RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITÉ DJILLALI LIABES SIDI BEL ABBES
FACULTÉ GÉNIE ÉLECTRIQUE
DÉPARTEMENT D'ÉLECTRONIQUE

THESE DE DOCTORAT

POUR OBTENIR LE GRADE DE

DOCTEUR EN SCIENCE

SPÉCIALITÉ : ÉLECTRONIQUE

OPTION : TRAITEMENT D'IMAGES & ARCHITECTURE PARALLÈLE

PRÉSENTÉE PAR :
ZOUAOUI CHAKIB MUSTAPHA ANOUAR

INTITULÉ :

Conception et implémentation d'une librairie de conteneurs réguliers méta-programmés pour les architectures parallèles, applicable aux systèmes numériques multidimensionnels

---

SOUTENUE LE : ....., DEVANT LE JURY COMPOSÉ DE :

Pr BOUKHELIF Aoued, (UDL-SBA)	: Président
Pr TALEB Nasreddine, (UDL-SBA)	: Directeur
Pr BENSLIMANE Sidi Mohamed, (ESI-SBA)	: Examinateur
Dr AMAR BENSABER Djamel, (ESI-SBA)	: Examinateur

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

À

la mémoire de mon défunt père,

ma mère,

mon frère Amine et mes sœurs Lynda et Nawel,

ma femme Fatima et mon fils Mohamed Ziyad,

mes nièces Lamia , Imene , Merieme et Sarah

mon beau-père,

ma belle-mère,

ma belle-soeur Yamina et mes beau-frères Arslan et Fouad,

mes beau-frères Nourddine et Mohamed,

mon directeur de thèse Pr Taleb,

et à la mémoire de mon encadreur en magistère, Pr. Brahimi née Menazla.

REMERCIEMENTS

A l'issue de la rédaction de cette recherche, je suis convaincu que la thèse est loin d'être un travail solitaire. En effet, je n'aurais jamais pu réaliser ce travail doctoral sans le soutien d'un grand nombre de personnes dont la générosité, la bonne humeur et l'intérêt manifestés à l'égard de ma recherche ont été d'un réel soutien.

En premier lieu, je tiens à remercier mon directeur de thèse, le professeur TALEB Nasreddine, pour la confiance qu'il m'a accordé en acceptant d'encadrer ce travail doctoral, pour ses multiples conseils ; sa grande disponibilité, ses qualités humaines et pour toutes les heures qu'il a consacrées à diriger cette recherche.

Mes remerciements vont également aux professeurs Aoued BOUKELIF ,Sidi Mohammed BENSLIMANE et au docteur Djamel AMAR BENSABER pour avoir accepté de participer à ce jury de thèse.

Je tiens également à remercier le professeur Ahmed LEHIRECH pour sa très grande ouverture d'esprit, de m'avoir prodigué maints conseils. De même, je suis particulièrement reconnaissant envers mon frère Amine Zouaoui de l'intérêt qu'il a manifesté à l'égard de cette recherche et ses nombreuses contributions.

Je souhaiterais aussi adresser ma gratitude à mes collègues du laboratoire RCAM, en particulier au docteur Miloud CHIKH EL MEZOUAR , aux professeurs Maachou ANNANI , Zouaoui CHAMA, Zoubir MAHDJOUB , Nacer BOUNOUA , Kamel BELOULATA, et messieurs Abbes BERAHEL et Kadour BEENKHALOUK) ,à mes autres collègues du département d'éléctronique en particulier (au professeur MALIKA KANDOUSSI, aux docteurs Azziz JOTI, Rachida MELIANI, Mohamed KHADRAOUI, Morad Medless, à monssieur Rabii Naiimi) et du département de télécommunication en particulier (aux professeurs Fouzi BELKHOUDJA, Ali DJEBARRI, Ahmed Djebbar , au docteur Abdel Krim Ghaz,et à monssieur Sidi Mohamed SEKKAL).

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Le langage Open Computing (OpenCL) a été proposé comme un standard ouvert par le groupe Khronos [11, 13, 20, 23, 56, 67, 68, 70]. Ce langage est conçu dans l'esprit de servir de socle unifié pour la programmation parallèle hétérogène sur des système à mémoire partagée, ciblant une large variété de dispositifs potentiellement parallèle (CPU,GPU et DSP) [37, 38, 69, 77, 78]. L'objectif adressé par le modèle unifié pour la programmation parallèle positionne OpenCL à la lisière de deux grands paradigmes de programmation parallèle CPU et GPU.
OpenCL s'articule autour de deux types de programmes [54] :.

1. Le premier type est un orchestrateur de programme généralement séquentiel appelé le programme hôte, qui implique le processeur hôte (CPU). Le programme hôte est essentiellement composé d'appels de fonction OpenCL (API).

2. Alors que le second type de programme comprend une ou plusieurs fonctions Kernel qui sont fondamentalement parallèles et écrites dans un langage à la sémantique statique du sous-ensemble C avec des extensions parallèles pour les versions d'OpenCL 1.1, 1.2 et 2.0 [67, 68, 70] , tandis que les versions plus récentes (OpenCL 2.0 et 2.1) implémentent les deux sous-ensembles C et C ++ [11, 13, 20, 23, 56]. Les fonctions C kernel permettent deux types d'arguments : un opérande scalaire et un pointeur unidimensionnel sur une collection de données contiguës (les éléments de la collection de données sont stockés dans un espace d'adresses virtuelles contigu). Le noyau C ++ 14 est compatible avec de nouveaux arguments, tels que les conteneurs POD

hérités du C, qui sont envoyés par valeur ou par référence.

Plus récemment et pour améliorer la productivité d'OpenCL, le groupe Khronos a ratifié SYCL, un modèle de programmation de haut niveau pour OpenCL à source unique basé sur le nouveau standard C ++ 14. SYCL propose une classe Buffer prévue pour le transfert des données entre le programme hôte et les API OpenCL.
La classe Buffer définit un espace de stockage contigu partagé, qui peut être à une, à deux ou à trois dimensions. La classe Buffer est un modèle qui expose de nombreux constructeurs de surcharge (pointeur unidimensionnel hôte sur une collection de données contiguës, des itérateurs unidimensionnels délimités par begin() et end(), ou par l'utilisation de son propre allocateur mémoire [43, 80]).
Bien que les tableaux multidimensionnels sont les structures de données les plus importantes dans les programmes impératifs, en particulier dans les applications de simulation scientifique, presque tous les calculs sont effectués sur des tableaux multidimensionnels. OpenCL ne fournit pas de conteneur de données multidimensionnel. En effet, les versions d'OpenCL nouvellement prévues (2.1 et 2.2) ne mettent en œuvre que des parties de la spécification std : : array unidimensionnelle telle que décrite dans le chapitre standard C ++ 14 [23.3.2], alors que la nouvelle ratification SYCL propose $vec < Typename$ T, int $dims >$, qui est un conteneur unidimensionnel non redimensionnable qui compile vers le vecteur intégré (une structure qui enveloppe 2, 4, 8 ou 16 types scalaires pris en charge dans le code du périphérique OpenCL) .
Dans un contexte plus général, le problème de la conception et de la mise en œuvre de conteneurs de données multidimensionnels dynamiques n'est pas spécifique à OpenCL ; Il concerne aussi les différentes normes du langage C ++ [51, 59].
Ainsi, en l'absence de conteneurs multidimensionnels efficaces fournis par OpenCL, les programmeurs OpenCL définissent leurs propres implémentations de pseudo-array en utilisant des méthodes d'accès qui émulent le comportement de tableaux multidimensionnels,

alors que les programmeurs C/C++ ont tendance à utiliser les conteneurs fournis par les bibliothèques boost : : Multiaray ou std : : vector de la STL, toutes deux connues pour leur niveau élevé d'abstraction au détriment de la performance, de l'interopérabilité et de la compatibilité entre plates-formes.
Ce manque constant de représentation dimensionnelle et l'absence de structures de données multidimensionnelles adaptées pour OpenCL sont au cœur de nos motivations.
Dans cette thèse, nous présentons CL_ARRAY, une nouvelle bibliothèque de conteneurs multidimensionnels qui garantit la contigüité des éléments, composée de trois types de conteneurs de données multidimensionnels : CL_ARRAY : :STACK, CL_ARRAY : : HEAP, et CL_ARRAY : :OCL.
Ces conteneurs sont tous dotés de méthodes d'accès C++ standard (par opérateur subscript []) afin d'assurer l'interopérabilité avec les plates-formes parallèles les plus populaires telles que OpenMP [5], MPI [3], C ++ AMP [65] et Intel TBB [44]).

1.2 Travaux connexes

Les conteneurs de données sont essentiels pour toute spécification de langage de programmation de haut niveau [84] ; cependant, l'absence de conteneurs STL dédiés à la programmation parallèle constitut un domaine de recherche de niche qui succite l'intérêt de bon nombre de chercheurs, qui ont mis en œuvre avec un relatif succès des langage intégré aux compilateurs C++ dédiés (Domain Enbeded Specific Language) intégrant des tableaux multidimensionnels.

- Kokkos [41] est un DSEL intégré pour C ++ qui fournit un modèle de programmation basé sur les tableaux (Array Based Programming) construit autour de données parallèles. Kokkos permet d'exécuter des codes noyaux portatable sur les périphé-

riques parallèles multicore (multi-backend) en utilisant des "access patterns " pour obtenir des temps de lecture et d'ecriture optimisés. Cependant, les tableaux multi-dimensionnels proposés par Kokkos sont des vues métaprogrammées multidimensionnelles qui limitent la dimensionalité à trois dimensions, ils sont conçus pour être utilisés directement par les dispositifs parallèles (CPU et GPU). Ce choix limite d'une part la taille des conteneurs à la taille de la mémoire disponible, et d'autre part, l'interopérabilité entre les conteneurs Kokkos et les autres DSELs qui intègrent des tableaux multidimensionnels.

- Dans [51], une extension de la bibliothèque UPC ++ [96] , une bibliothèque C/C ++ basée sur le partitionnement d'espace d'accès (PGAS), permet des vues multidimensionnelles limitées à trois dimensions. Contrairement à Kokkos, PGAS permet de créer des zones mémoires dans la mémoire locale. Elle offre également plusieurs opérations associées à des vues (transformations multidimentionnelles,hachage, etc.).

- Les conteneurs intelligents de SKEPE [27], un DESL de squelette métaprogrammé C ++ prenant en charge plusieurs backends, sont particulièrement connus pour optimiser la gestion de la mémoire et les transferts entre des systèmes basés sur CPU et multi-GPU. Ces conteneurs intelligents utilisent une interface similaire aux bibliothèques C ++ STL. Les auteurs ont démontré une optimisation en temps et des économies conséquentes en bande passante. Cependant, ces conteneurs intelligents ne peuvent être utilisés que dans le contexte du DSL, et ils sont limités à deux dimensions.

- ArrayFire [93] est un autre exemple de bibliothèque de programmation parallèle multi-language (C, C ++, C# , java etc.) et multi-backend (cuda et OpenCL). Le

modèle de programmation de ArrayFire repose sur l'utilisation de fonctions de calcul à grande échelle (centaines de fonctions de différents champs mathématiques et scientifiques) appliquées sur des tableaux multidimensionnels. Le code utilisateur est traduit lors de la compilation en programmes Kernels parallèles. Cependant, les tableaux multidimensionnels de la bibliothèque ArrayFire sont limités à trois dimensions, et ils ne peuvent stocker que des données numériques. En outre, les tableaux ArrayFire ne peuvent pas être exploités en dehors de la DESL.

- STAPL est un DESL métaprogramé C ++ destiné à la programmation parallèle avec un très haut niveau d'abstraction [86]. STAPL a été créé conformement aux spécifications du STL standard C++ . Chaque composant STAPL (Containers, algorithmes et itérateurs) a été spécialisé et augmenté à des fins d'éxcutions parallèles sur des dispositifs à mémoire partagée et|ou distribuée . Pvector, est un conteneur parallèle proposé par STAPL, il fournit une vue d'objet "partagée" et il est toujours associé à un itérateur parallèle appelé "View" qui permet la segmentation des conteneurs en sections et sous-sections. Chaque section ou sous-section peut être exécutée indépendamment sur un dispositif parallèle. Cependant, les conteneurs STAPL restent incompatibles avec OpenCL .

- D'autres travaux ont ciblé des compilateurs autres que C ++ mais qui sont largement utilisés par les développeurs et la communauté scientifique. Par exemple, les bibliothèques NumPy Array [87] et PyCUDA / PyOpenCL [19] supportent les tableaux multidimensionnels et sont conçues pour le langage de programmation Python. En particulier, PyCUDA / PyOpenCL sont spécialement conçus pour être utilisées avec un GPU et supportent plusieurs backends (CUDA et OpenCL). Haskell repa [52] est une autre bibliothèque connue pour son langage formel (Haskell) qui supporte des tableaux multidimensionnels parallèles.

1.3 Contributions

la conception d'une librairie, générique , performante et exempte de toute ambigüité sémantique à la vue des utilisateurs, est une composante essentielle pour la réussite de toute librairie [74], c'est encore plus vrai, lorsqu'il s'agit de la conception d'une sémantique destinée à la génération et la manipulation des conteneurs de données multidimensionnels. Dans cette thèse nous proposons une refonte des aspects conceptuels classiquement suivis par les concepteurs de librairies de conteneurs, et nos principales contributions s'articulent autour de propositions suivantes :

1. Un nouveau formalisme pour représenter la nature des systèmes multidimentionnels, une syntaxe et une sémantique unifiée pour la génération et manipulation des conteneurs multidimensionnels statiques et dynamiques.

2. CL_ARRAY : :STACK, alloué dans la mémoire pile, est un conteneur de données de métaprogrammé équivalent au conteneur statique std : :array avec un sémantique multidimensionnelle, utilisable avec ou sans le framework OpenCL.

3. CL_ARRAY : :HEAP, alloué dans la mémoire dynamique (mémoire de tas), est un conteneur métaprogrammé pour les tableaux dynamiques multidimensionnels utilisables avec ou sans le framework OpenCL.

4. CL_ARRAY : :OCL, alloué dans la mémoire Heap et la mémoire des dispositifs OpenCL, utilisables uniquement avec le framework OpenCL et conçu pour accélérer les transferts de données entre la mémoire hôte et les dispositifs OpenCL.

5. Les structures CL_ARRAY : :HEAP et CL_ARRAY : :OCL sont conçues avec de nouveaux algorithmes basés sur une représentation par pointers hiérarchiques multi-niveaux qui forment une structure arborescente. Chaque dimension est représentée comme un noeud de l'arbre, dont les indices d'une dimension sont liés aux

sous-arbres de la dimension inférieure. Cette approche produit des tableaux multi-dimensionnels de style C / C ++ avec une légère surcharge mémoire par rapport aux pseudo-arrays ; mais avec un niveau de performance très similaire aux pseudo-array sans restreindre la dimensionnalité et sans sacrifier la généricité et/ou la portabilité des programmes.

6. De nouveaux algorithmes pour le redimensionnement, l'insertion et la suppression (le retrait) adaptés aux structures proposées.

7. OCL_ARRAY_VIEW, une structure multifonction fournissant aux utilisateurs des transformations multidimensionnelles telles que ARRAY_VIEW du C ++ AMP. Contrairement aux vues multidimentionnelles du C++ AMP , OCL_ARRAY_VIEW propose l'accès par un opérateur standard C / C ++, une nouvelle sémantique d'itérateurs multidimensionnels ainsi qu'un accès sécurisé par pointeurs unidimensionnels requis par toutes les versions d'OpenCL (1.x et 2.X), permettant aux conteneurs CL_ARRAY d'interagir avec la bibliothèque STL, les API OpenCL et les objets buffeur fournis par SYCL, qui nécessitent des pointeurs unidimensionnels ou des itérateurs begin()/ end().

1.4 Organisation du manuscrit

La présentation de nos travaux se déroule de la manière suivante :

- Nous présentons dans le chapitre 2 un état de l'art sur les architectures et les plateformes parallèles les plus populaires auprès de la communauté des programmeurs parallèles, à savoir OpenMP, Intel TBB, MPI et CUDA.

- Nous mettons l'accent dans le chapitre 3 sur les aspects techniques les plus importants afin d'apporter de la clareté aux lecteurs quand au fonctionnement , le modèle programmatique , l'évolution et les projets majeurs échafaudés autour d'OpenCL.

- Le chapitre 4 est destiné à fournir une vue d'ensemble bien que succincte du paradigme de la métaprogrammation. Ce chapitre aborde les principes programmatiques spécifiques à la métaprogrammation , tels que les politiques , les traits , les pointeurs intelligents ou encore les expressions template. Il introduit également les conteneurs de données proposés par la librairie STL.

- Dans le chapitre 5 nous introduisons les aspects conceptuels de la librairie CL_ARRAY, à travers de laquelle nous explicitons nos propositions sémantiques pour la représentation de la dimensionnalité et des transformations multidimensionnelles prévues pour formaliser le modèle d'exécution d'OpenCL. Nous présentons également les algorithmes majeurs que nous étayons par leur complexité algorithmique ; concus pour la génération , la libération , le redimensionnement , la suppression et l'insertion des éléments.

- Le chapitre 6 est prévu pour présenter les techniques programmatiques utilisées pour l'implémentation de la librairie CL_ARRAY, il présente également des exemples concrets d'utilisation et des évaluations des performances des conteneurs en termes d'adaptation dans des applications à forte dimensionnalité, de leurs capacité à optimiser les transferts de données entre l'hote et les dispositifs OpenCL et enfin des comparatifs avec les librairies de conteneurs les plus utilisés par la communauté des développeurs (std : :vector,Biltz++ et boost : :multiarray) à l'appui d'applications naïve d'algèbre multilinéaire et de convolution d'images.

- Le chapitre 7 dresse un bilan de nos travaux et avance des pistes nouvelles pour le futur.

CHAPITRE 2

ARCHITECTURES ET ENVIRONNEMENTS PARALLÈLES

2.1 introduction

De nos jours, la plupart des unités de traitements modernes aussi bien spécialisés qu'à usage généralistes supportent le parallélisme [54]. À l'origine, l'écriture d'applications ciblant des architectures parallèles était non seulement réservée à des fins de calcul de haute performance mais aussi accessible à une niche d'experts, de chercheurs et de spécialistes de renoms, tandis que la majorité des programmeurs ciblaient exclusivement les architectures séquentielles et profitaient des améliorations des performances grâce à l'augmentation rapide des fréquences de fonctionnement des unités de traitements. Cependant, l'augmentation de la fréquence des processeurs a atteint, il y a quelques années, un seuil infranchissable causé par des limitations physiques. Pour davantage de performances, certaines formes de parallélisme implicite qui échappaient complètement au contrôle des programmeurs ont été introduites dans les processeurs pour en améliorer l'efficacité (pipelining des instructions, vectorisation, et exécutions multi-cœurs)[32]. Désormais, et par acquis de conscience que le parallélisme implicite ne peut exploiter à lui seul tout le potentiel parallèle des unités de traitement modernes, le parallélisme contrôlable par les programmeurs est omni présent, des super-calculateurs jusqu'aux processeurs en passant par les appareils embarqués tels que les téléphones portables. Il advient et dans l'état actuel qu'une exploitation explicite du potentiel parallèle est l'unique moyen permettant d'exploiter d'une manière rationnelle le potentiel offert par le matériel [75]. De plus, et bien que les exigences des applications à usage générale (jeux électronique, bases de données, applications WEB..etc) sont moindres que celles imposées par le calcul scientifique, il n'en demeure pas moins, que la programmation parallèle introduit en tout état de lieu de

nouvelles difficultés de programmation. Il est en particulier question de palier tout risque lié à l'incohérence causée par les conditions de courses et les inter-blocages . Ce chapitre est consacré à l'étude des environnements parallèles les plus répandus auprès de la communauté des programmeurs parallèles et des chercheurs spécialisés.

2.2 Modèles d'exécution

2.2.1 Taxonomie de Flynn

Sur une machine séquentielle, l'unique modèle d'exécution est celui de la machine de Von Neumann : le flot d'instructions (unique) est exécuté au rythme d'une instruction par cycle d'horloge, chaque instruction intervenant sur une donnée unique (scalaire). Dans le domaine du calcul parallèle, une application peut correspondre à plusieurs flots d'instructions différents, sur un unique ensemble de données ou sur des flots de données différents. La classification proposée par Michael J. Flynn [54, 63] se base sur ces distinctions ; elle met en évidence quatre types de modèles d'exécution.

2.2.1.1 SISD : Single Instruction stream - Single Data stream

Ce modèle d'exécution est celui du monde séquentiel pour des ordinateurs à processeur scalaire. Un programme est un flot d'instructions sur un flot de données, Les machines monoprocesseur pipelinées ou super-scalaires entrent également dans cette catégorie, en relevant plus du traitement simultané que de l'exécution concurrente.

2.2.1.2 SIMD : Single Instruction stream - Multiple Data stream

Dans ce modèle d'exécution, le même flot d'instructions est exécuté de façon concurrente par différents processeurs, chacun sur un flot de données. Les instructions s'exécutent de façon synchrone.

2.2.1.3 MISD : Multiple Instruction stream - Single Data stream

Le modèle d'exécution MISD correspond à des architectures composées de plusieurs unités fonctionnelles qui travaillent chacune leur tour sur un même jeu de données, de façon synchrone. C'est typiquement le cas des tableaux systoliques de processeurs (chaque processeur correspond à une unité fonctionnelle). On peut se représenter le transfert des données comme si elles étaient "aspirées" d'une unité fonctionnelle à l'autre (l'image est celle du cœur, qui pompe le sang). Un tel modèle d'exécution est adapté à des applications très spécifiques qui peuvent être issues de différents domaines, par exemple :

- dans le domaine du traitement d'image, il permet d'exécuter plusieurs traitements consécutifs sur une image ;
- des machines à trier ont été conçues sur ce modèle : les unités fonctionnelles, rudimentaires, absorbent des valeurs qu'elles transmettent à l'un de leurs voisins selon leur ordre ; on implémente également ainsi des files de priorités : à chaque étape, la valeur entrée transite vers sa position et la plus petite valeur est extraite .

2.2.1.4 MIMD : Multiple Instruction stream - Multiple Data stream

Il s'agit du modèle le plus large pour exprimer le parallélisme : plusieurs processeurs sont nécessaires, et chacun exécute son propre flot d'instructions, de façon autonome et sur un flot de données qui lui est propre. Ce modèle offre le plus de possibilités, en permettant de concevoir des applications constituées de tâches, dépendantes ou non, exécutables en parallèles ; mais les communications et/ou synchronisations entre les processeurs doivent être gérées explicitement, et il faut équilibrer la charge répartie sur les processeurs.

2.2.2 Modèles d'architecture relevant du MIMD - classifications de Duncan et Young

Duncan a proposé des modifications à la taxonomie de Flynn pour appréhender les nouveaux types des architectures selon les différentes méthodes d'accès physique à la mémoire [54, 63]. Selon sa classification il y a deux classes génériques :

- architectures à mémoire partagée.
- architectures à mémoire distribuée.

Les modèles à mémoire partagée donnent une vue de la mémoire selon un unique espace d'adressage : il s'agit de la vue utilisateurs, et les applications qu'ils développent. Quant aux modèles à mémoire distribuée, ils ne prévoient pas que la mémoire puisse être utilisée de façon partagée : dans ce modèle, chaque processeur dispose nécessairement de sa mémoire localement. En bas niveau, les échanges entre les processeurs (communication de données, synchronisation) doivent alors forcément s'effectuer par échange de messages [22, 75]. Le tableau I.1 présente d'une manière synthétique les avantages et les inconvénients de ces deux types d'architecture [64]. En partant du type d'interconnexion, Young et al [54, 64]. proposent une classification plus fine des architectures selon l'accès physique à la mémoire :

Tableau 2.I – Avantages et inconvénients des architectures à mémoire partagée et distribuée

	Mémoire partagée	**Mémoire distribuée**
Avantages	• l'espace d'adressage global permet un accès mémoire facile via un modèle de programmation intuitif. • le partage des données se fait de manière rapide et uniforme grâce au rapprochement de la mémoire par rapport au processeur.	• passage à l'échelle facile avec l'augmentation du nombre de processeurs • la taille mémoire augmente proportionnellement avec le nombre des processeurs • chaque processeur a accès à sa propre mémoire sans coût supplémentaire pour assurer la cohérence des caches
Inconvénients	• passage à l'échelle difficile car l'ajout des processeurs peut augmenter géométriquement le trafic sur les interconnexions mémoire-processeurs • responsabilité de l'utilisateur pour faire la synchronisation • coût considérable pour la conception de machine avec un nombre conséquent des processeurs	• difficile à programmer : l'utilisateur doit se charger de toutes les communications inter-processeurs • temps d'accès non-uniforme • difficile de modéliser la répartition de données

2.3 Les langages parallèles

2.3.1 MPI (The Message Passing Interface)

Le modèle de programmation MP était initialement conçu pour fonctionner sur un réseau de calculateurs. Il est le modèle le plus populaire et le plus mature qui permet l'écriture d'applications sur des systèmes à mémoire distribuée, il a pris son essor depuis les années 1990 dans l'optique de formaliser et de synthétiser le parallélisme sur grilles de calcul et plus tard pour les systèmes HPC. Ce modèle est basé essentiel lement sur deux concepts de base [39, 64, 92] :

- Le premier concept introduit la notion de communicator [64], qui définit un groupe de processus qui ont la capacité de communiquer les uns avec les autres. Dans ce groupe de processus, chacun reçoit un rang unique, et ils communiquent explicitement les uns avec les autres en indiquant leurs rangs respectifs [3, 39, 75].

- Le second concept tente à formaliser les échanges interprocessus, qui reposent exclusivement sur les opérations d'envoi et de réception de messages entre les processus d'un même groupe . Un processus peut donc envoyer un message à un autre processus en fournissant son rang et une étiquette unique pour identifier le message. Le processus récepteur peut dès lors affirmer la réception du dit message, manipuler les données en conséquence. Ces communications qui impliquent un unique expéditeur et un unique destinataire sont appelées communications point à point. Toutefois, et si le besoin se fait sentir d'envoyer le même message à tous les processus impliqués dans un même groupe, MPI propose un mécanisme dit d'envoi collectif qui permet d'optimiser ces échanges et éviter ainsi à la fois une surcharge inutile du code (bannir tout envoi multiple d'un même message point à point) et palier toute congestion du réseau [39, 75].

2.3.1.1 Model de programmation MPI

A l'origine, [54, 64] a été conçu pour les architectures distribuées de mémoire, qui ont été de plus en plus populaires à cette époque (années 1980 - début des années 1990). Comme les tendances de l'architecture ont changé, les développeurs MPI ont adapté leurs bibliothèques pour gérer la majorité des types d'architectures de mémoire sous-jacents de façon transparente. Ils ont également adapté / développé des moyens de manipulation différentes (interconnexions et des protocoles). Aujourd'hui, MPI fonctionne sur pratiquement toute plate-forme matérielle [3] :

- Mémoire distribuée
- Mémoire partagée
- Hybride

Cependant le modèle de programmation reste clairement un modèle de mémoire distribuée [64], quelle que soit l'architecture physique sous-jacente de la machine. Tout le parallélisme est explicite : le programmeur est responsable d'identifier correctement le parallélisme et la mise en œuvre des algorithmes parallèles utilisant des constructions MPI.

2.3.1.2 Schéma général d'un programme MPI

rang <- identificateur du processus en cours par rapport à un communicateur
Si (rang = identificateur_spécifique) **Alors** faire un traitement
Sinon faire un autre traitement

2.3.1.3 Communicateurs et groupes

Dans un communicateur [54, 64], chaque processus a son propre identifiant entier unique attribué par le système lorsque le processus initialise. Un rang est parfois aussi appelé

«ID de tâche ». Les rangs sont contigus et commencent à zéro. Un rang est utilisé par le programmeur pour spécifier la source et la destination des messages et il est Souvent utilisé conditionnellement par l'application pour contrôler l'exécution du programme (si rang = 0 faire / si rang = 1 faire).

2.3.1.4 Algorithme de gestion des groupes

Cependant, ce modèle de programmation offre une prise en charge très limitée du parallélisme à mémoire partagée, cette limitation est généralement contournée par la conception de programme hybride : MPI est utilisé pour le parallélisme à gros grain entre les différents nœuds de calcul, où chaque processus MPI est à son tour parallélisé à travers une plateforme dédiée au parallélisme CPU (par exemple en utilisant OpenMP [30]

2.3.2 OpenMP

OpenMP [5, 26, 30] (Open Multi-Processing) est modèle de programmation pour le calcul parallèle sur architecture à mémoire partagée. OpenMP est supporté sur de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Il s'agit d'un style de programmation basé sur des directives, il permet de produire des applications parallèles avec un minimum d'effort à petite granularité tout en restant très proche du code séquentiel. Cependant OpenMP n'offre que peu voire aucune garantie pour une exécution parallèle. En effet, le compilateur essayera autant que possible de produire un code parallèle implicite conformément aux directives introduites par les programmeurs, mais si il se trouve dans l'incapacité de parallèliser le code, il procédera à la sérialisation de ce dernier (production d'une version séquentielle du programme).

2.3.2.1 Model de programmation

2.3.2.1.1 Parallélisme Fork-Join Dans ce modèle, le Thread Master d'OpenMP produit implicitement une gropue de threads incrémentielle jusqu'à ce que les objectifs de rendement soient atteints [26, 30, 62, 64].

2.3.2.1.2 OpenMP API Les APIs OpenMP sont classées et organisées comme suit :

- Directives pour expliciter le parallélisme, les synchronisations et le statut des données (privées, partagées...).
- Bibliothèque pour des fonctionnalités spécifiques (informations dynamiques, actions sur le runtime...).
- Variables d'environnement pour controler le comportement parallèle du programme OpenMP (nombre de threads, stratégies d'ordonnancement...)

2.3.2.2 Modèle mémoire OpenMP

Le modèle mémoire OpenMP, implique que [5, 26] :

1. Les transferts de données sont transparents pour le programmeur.
2. Tous les threads ont accès à la même mémoire partagée [26]
3. Les données partagées sont accessibles par tous les threads
4. les données privées sont accessibles seulement par les threads correspondants.

2.3.2.3 La synchronisation

OpenMP [5] met à la disposition des programmeurs des directives de synchronisation mult-threads, quant au déroulement de l'exécution. Il s'agit de directives :

- ATOMIC
- CRITICAL
- BARRIER
- FLUSH

Les directives ATOMIC et CRITICAL permettent de sérialiser l'exécution des threads suivant immédiatement la directive, la différence entre les deux directives réside dans le fait que ATOMIC se base principalement sur les instructions atomiques des processeurs, trandis que CRITICAL englobe toute une partie de code [5, 62]. La directive BARRIER est une barrière de synchronisation : tout thread l'atteignant ne pourra progrésser avant que tous les threads arrivent à la barrière. La directive FLUSH (variable) permet de rafraîchir la variable partagée variable visible de tous les processeurs. Elle est d'autant plus utile que la mémoire d'une machine est hiérarchisée. Elle peut servir à synchroniser dans une boucle parallélisée avec OMP DO/for. La directive ORDERED impose au compilateur de respecter l'ordre des opérations à l'intérieur d'un nid de boucle. Bien qu'elle réduit considérabelement les degrés de parallèlisme, elle garantit un ordre d'exécution cohérent correspondant aux itérations de la boucle, conforme à une exécution séquentielle.

2.3.3 Intel TBB

Intel Threading Building Blocks [24, 44, 62], est une bibliothèque logicielle développée par la société Intel. Elle prend en charge le parallélisme basé sur un modèle d'attribution des tâches. Elle présente les caractéristiques suivantes :

- Bibliothèque de modèles supportant à la fois le parallélisme régulier et irrégulier.
- Un soutien direct pour une variété de modèles de parallèlisme structuré (MAP, fork–join, graphes de tâches, réduction, et pipelines).
- Des ordonnoceurs prêts à l'emploi pour l'équilibrage des charges de travail.
- Une collection de structures de données thread-safe (protégé des conditions de courses).
- Prise en charge des opérations atomiques et des allocations implicites de la mémoire.

Intel TBB est un DESL [44], et peut donc être utilisée avec tout compilateur supportant ISO C ++. TBB expose aux programmeurs un ensemble d'objets et de fonctions pour spécifier des blocs de code à exécuter en parallèle. A l'instar de la STL du C++ dans laquelle les interfaces sont spécifiées uniquement par des exigences sur les types de modèles, TBB suit une philosophie similaire. TBB repose donc sur des modèles et programmation générique avec peu d'hypothèses sur la nature des structures de données, ce qui maximise le potentiel de réutilisation des primitives TBB. TBB est basée sur une sémantique de tâches dans l'optique de réduire les frais généraux et de gérer plus efficacement les ressources . Les composants individuels de TBB peuvent également être utilisés avec d'autres modèles de programmation parallèle. Par exemple, Il n'est pas rare de voir l'allocateur de mémoire parallèle TBB utilisé avec des programmes OpenMP [64].

2.4 Langages spécifiques aux accélérateurs graphiques

Les deux langages propriétaires sont de plus en plus utilisés par les programmeurs : CUDA pour le constructeur des cartes graphiques NVIDIA [64, 75] et C++ AMP [65] édité par Microsoft.

2.4.1 CUDA – NVIDIA

2.4.1.1 Model de programmation

2.4.1.1.1 Kernels C pour CUDA [7, 30, 64] étend C en permettant au programmeur de définir des fonctions C, appelées noyaux, elles sont exécutées N fois en parallèle par N différents threads CUDA, par opposition à une seule fois comme les fonctions régulières C. Un noyau est défini en utilisant le spécificateur de déclaration de __global__ , et le nombre de threads CUDA pour chaque appel est spécifié à l'aide d'une nouvelle syntaxe «<... »>.. Chacun des threads qui exécutent un noyau est donné un ID de thread unique qui est accessible dans le noyau grâce à la variable intégrée threadIdx. Comme l'illustre l'exemple de listing 2.1, le résultat de l'addition de deux vecteurs A et B de taille N est stocké dans le vecteur C :

```
1  // definition du noyau
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4  int i = threadIdx.x;
5  C[i] = A[i] + B[i];
6  }
7  int main()
8  ...
9  // Kernel invocation
10 VecAdd<<<1, N>>>(A, B, C);
11 }
```

Listing 2.1 – CUDA Kernel adition de deux vecteurs

Chacun des threads qui exécute VecAdd () effectue une addition par paire.

2.4.1.2 Hiérarchie de Mémoire

Threads CUDA [7] peuvent accéder à des données provenant de multiples espaces de mémoire lors de leur exécution, chaque thread a une mémoire locale privée. Chaque bloc de threads a une mémoire partagée visible pour tous les threads du bloc et avec la même durée de vie que le bloc. Enfin, tous les threads ont accès à la même mémoire globale. Il y a aussi deux espaces de mémoire en lecture seule supplémentaires accessibles par tous les threads : les espaces de mémoire constants et de texture. Les espaces globaux, constants, et la mémoire texture sont optimisés pour différents usages de la mémoire. La mémoire de texture offre également différents modes d'adressage, ainsi que le filtrage des données (pour certains formats de données spécifiques).

2.4.1.3 Hôte et Device

Le modèle de programmation CUDA [7, 30]suppose que les threads CUDA exécutent sur un dispositif physiquement séparé qui fonctionne comme un coprocesseur à l'hôte. C'est le cas, par exemple, lorsque les threads s'exécutent sur un processeur graphique alors que le reste du programme C est exécuté sur le processeur central. Le modèle de programmation CUDA suppose également que l'hôte et le dispositif maintiennent leur propre DRAM, appelée mémoire hôte et la mémoire de l'appareil, respectivement. Par conséquent, un programme gère les espaces globaux, constants. Cela inclut l'allocation de mémoire de l'appareil et dé allocation, ainsi que le transfert de données entre l'hôte et la mémoire de l'appareil.

2.4.2 C++ AMP

C++ AMP (C++ Accelerated Massive Parallelism) [65, 66] tente d'accélérer l'exécution du code C++ en tirant parti du matériel tel que le processeur graphique (GPU). Le mo-

dèle de programmation C++ AMP inclut des tableaux multidimensionnels comme illustré par le listing 2.2, l'indexation, le transfert de mémoire, la mosaïque et une bibliothèque de fonctions mathématiques. Afin d'améliorer les performances, il est toutefois possible d'utiliser les extensions du langage C++ AMP pour contrôler la façon dont les données sont déplacées de l'UC au GPU et inversement. Le modèle de programmation C ++ AMP [64–66] est pris en compte dans les fichiers d'en-tête suivants :

- <amp.h>
- <arm_math.h>
- <amp graphics.h>
- <amp court vectors.h>

```
#include <amp.h>
#include <iostream>
using namespace concurrency;
void AddArrays() {
int aCPP[] = {1, 2, 3, 4, 5};
int bCPP[] = {6, 7, 8, 9, 10};
int sumCPP[5] = {0, 0, 0, 0, 0};
array_view<int, 1> a(5, aCPP);
array_view<int, 1> b(5, bCPP);
array_view<int, 1> sum(5, sumCPP);
parallel_for_each(sum.extent,[=](index<1> idx) restrict(amp)
{
   for (int i = 0; i < 5; i++)
   {
      std::cout << sum[i] << "\n";
   }
}
```

Listing 2.2 – Adition de deux vecteurs avec C++ AMP

Le modèle de programmation C ++ AMP est contenu dans l'espace de noms "concurrency"et les espaces de noms imbriqués. Voici les types et les modèles les plus importants qui composent C ++ AMP :

- **Niveau d'indexation <amp.h>** : index<N>, extent<N>, tiled_extent<D0,D1,D2>, tiled_index<D0,D1,D2>.
- **Niveau de données <amp.h>** : array<T,N> , array_view<T,N>, array_view<const T,N> , copy ,copy_async .
- **Niveau d'exécution <amp.h>** : accelerator ,accelerator_view ,completion_future .
- **Niveau du noyau <amp.h>** : tile_barrier ,restrict() clause ,tile_static , fonctions atomiques .
- **Fonctions mathématiques <amp_math.h>** : fonctions mathématiques précises, fonctions mathématiques rapides.
- **Types de vecteurs court <amp_short_vectors.h>**.

2.5 Conclusion

Comme présenté par ce présent chapitre et dans le contexte bien particulier d'écriture de programmes parallèles, il existe de nombreux modèles de programmation. Parmi ces modèles de références, nous avons présenté MPI , un modèle de programmation destiné aux architectures distribuées, OpenMP pour le parallélisme multi-processeurs à mémoire partagée ou encore CUDA pour le parallélisme sur processeurs graphiques . Ces modèles de bas niveau permettent d'exploiter les capacités des machines parallèles. La contrepartie est qu'avec ces modèles il faut développer plusieurs versions d'une même application. Pour éviter l'effort multiple, il n'est pas rare que les concepteurs de logiciels acceptent des

concessions liées aux performances au profit de la portabilité des programmes. D'autre part et vu la complexité de l'écriture des programmes parallèles, les concepteurs de logiciels s'inquiètent aussi de leur productivité et de leur compétitivité. En effet, un juste équilibre entre l'exploitation quasi optimale des ressources parallèles et des temps raisonnables de développements de tests de pré et de post production est rarement atteint.

En guide de conclusion nous précisons également que OpenCL incarne l'une des tentatives les plus abouties qui a pour ambition de proposer un modèle de programmation parallèle unifié pour l'ensemble des architectures et des dispositifs parallèles. OpenCL est donc au cœur de notre étude et pour laquelle nous consacrons le chapitre 4, dans l'optique d'illustrer son fondement, son fonctionnement et son modèle de programmation.

CHAPITRE 3

LA PROGRAMMATION PARALLÈLE HÉTÉROGÈNE

3.1 Introduction

OpenCL est un standard industriel ouvert pour la programmation hétérogène ciblant en priorité les dispositifs parallèles à mémoire partagée CPU, GPU et DSP [16, 17, 21, 36, 38, 67–70, 78]. Il est plus qu'un language. En effet, OpenCL est un écosystème complet pour la programmation parallèle et comprend une riche bibliothèque d'APIs pour soutenir le développement de logiciels parallèls [38].
Non seulement les noyaux OpenCL peuvent fonctionner sur différents types de dispositifs, mais peuvent même être utilisés pour accélérer le traitement OpenGL ou Direct3D [38].
Ce présent chapitre présente les principes de base, sur lesquels l'écosystème s'articule, les modèles programmatiques , la sémantique des exécutions et des dépendances , l'évolution et les projets connexes construits autour d'OpenCL.

3.2 Le modèle OpenCL

L'objectif d'OpenCL est de fournir un modèle programmatique parallèles efficace tout en offrant une abstraction au fonctionnement du matériel sous-jacent , aux plateformes logicielles , à la gestion de la mémoire , au paradigme de programmation à mémoire partagée et à l'exécution parallèle, OpenCL est basé par conséquence sur les quatre modèles d'abstractions suivants [38, 69] :

3.2.1 Modèle de plate-forme

Le modèle de la plate-forme d'OpenCL est prévu pour fournir une abstraction totale aux plateformes matérielles sous-jacentes. Le modèle d'abstraction est basé sur un programme hôte (ordinateur équipé d'un processeur exécutant un système d'exploitation standard) qui contrôle et ordonne les exécutions parallèles sur un ou plusieurs périphériques OpenCL , qui peuvent etre des processeurs spécialisés GPU ou DSP, ou un ou plusieurs processeurs à usage général d'une technologie multi-core.
Pour simplifier la programmation hétérogène , un dispositif OpenCL est présenté au programmeur autant qu'une collection d'une ou plusieurs unités de calcul.
Une unité de calcul est composée d'un ou de plusieurs éléments de traitement SIMD (Single Instruction Multiple DATA) ou SPMD (single program multiple data) [38, 69, 77].
Les instructions SPMD sont généralement associées à des dispositifs à usage général (Exemple : processeurs X86), alors que les instructions SIMD s'exécutent sur un processeur vectoriel tel qu'un GPU [38, 69, 77].

3.2.2 Modèle d'exécution

Le modèle d'exécution OpenCL comprend deux composantes : noyaux (Kernels) et programmes hôte (Host). les fonctions Kernels sont l'unité de base di code exécutable qui fonctionne sur un ou plusieurs dispositifs d'OpenCL. Elles restituent la sémantique d'un sous-ensemble de fonctions C statiques avec une sémantique parallèle.
Le programme hôte est tout le temps exécuté sur le système hôte, il définit essentiellement le contexte dans lequel les dispositifs seront utilisés, dans le cas précis où plusieurs taches (fonctions noyaux) sont définies par l'utilisateur, le programmeur se doit de gérer leur ordonnancement dans des files d'attente et opter pour une gestion intrinsèque de ces dernières [38, 69, 77].
OpenCL exploite le calcul parallèle sur les dispositifs de calcul en définissant des modèles

abstraits dans un espace d'index à N-dimensions. Quand un noyau est en file d'attente, un espace d'index est défini. Chaque élément indépendant dans cet espace d'index est appelé un élément de travail. Chaque élément de travail exécute la même fonction noyau, mais sur des données différentes (parallélisme des données). La dimensionnalité de l'espace des indexs est définie par le programmeur, qui peut selon le cas opter pour un espace d'index à une , à deux ou à trois dimensions [38, 69, 77].
OpenCL impose également le regroupement des éléments de travail (work-items) en groupes de travail (work-groups). La taille de chaque groupe de travail est définie par son propre espace d'index local.
Tous les éléments de travail dans le même groupe de travail sont exécutés sur le même dispositif. Permettant ainsi aux éléments de travail à partager la mémoire locale.
Les éléments de travail globaux sont indépendants et ne peuvent pas être synchronisé. La synchronisation est autorisée uniquement entre les éléments de travail du même groupe de travail [38, 69, 77].

3.2.2.1 files d'attentes

La file d'attente de commande (command queue) est le mécanisme par lequel le programme hôte orchestre les exécutions parallèles sur les dispositifs OpenCL. Lorsqu'un kernel est en file d'attente, le dispositif exécute la fonction correspondante selon les deux règles suivantes [38, 69, 77] :

1. Chaque périphérique est associé à une file d'attente
2. Toutes les commandes parallèles sont passées aux périphériques par le biais de sa file d'attente.

3.2.2.2 Le contexte

Le contexte est au cœur de toutes les applications OpenCL qui fournit un conteneur objet de dispositifs parallèles associés à des noyaux OpenCL englobant des attributs mémoire (des tampons mémoires à une dimension et des objets mémoires images), des files d'attente de commande fournissant une interface entre le contexte et le dispositif parallèle [38, 69, 77]).
Outre sa capacité à centraliser la gestion des exécutions parallèles, le contexte fédère la communication avec et entre les dispositifs et les API OpenCL. A cet effet, un objet mémoire est associé à un contexte, mais sera mis à jour par un le dispositif.
Les API OpenCL garantissent que tous les dispositifs, dans le même contexte, procéderont à des mises à jours ordonnancées avec des points de synchronisation définis par l'utilisateur.
Dans le cas ou un programmeur souhaite répartir les traitements sur plusieurs dispositifs (par exemple : CPU et GPU), il sera contraint de créer et d'associer un contexte a chacun des dispositifs et d'envisager une synchronisation entre les contextes par la définition d'un ou plusieurs points de synchronisation [38, 69, 77].

3.2.3 Modèle de mémoire

Le modèle mémoire dicte la manière avec laquelle les noyaux interagissent avec l'hôte, et comment ils interagissent avec d'autres noyaux. OpenCL définit deux types d'objets de mémoire :

- **Objet mémoire tampon :** est un bloc contigu de mémoire mis à la disposition des noyaux. Un programmeur peut cartographier les structures de données sur ce tampon et accéder à la mémoire tampon par l'intermédiaire des pointeurs unidimensionnels [38, 69, 77].

- **L'Objet image :** est un objet mémoire destiné exclusivement au stockage optimisé des images à deux et à rois dimensions. Pour que les mises en œuvre OpenCL aient la flexibilité et la liberté de personnaliser le format d'image. l'objet de mémoire image, est proposé avec une sémantique d'objet opaque.

Formellement , le modèle de mémoire OpenCL définit cinq zones de mémoire distinctes :

- **Mémoire de l'hôte** : Cette zone mémoire est visible uniquement par le programme hôte, OpenCL définit seulement comment la mémoire hôte interagit avec les objets et les APIs OpenCL.
- **Mémoire globale** : Cette région de mémoire permet un accès en lecture / écriture à tous les éléments de travail communs aux groupes de travail. Les éléments de travail peuvent ainsi lire etlou écrire tout élément d'un objet résidant dans la mémoire globale. La lecture et l'écriture dans la mémoire globale peuvent être mis en cache en fonction des capacités du dispositif.
- **Mémoire constante** :Cette zone mémoire est allouée dans la mémoire globale mais demeure constante pendant l'exécution de la fonction kernel. Les éléments de travail ont un accès à la mémoire de travail en mode lecture seule.
- **Mémoire locale** : Cette zone mémoire est locale à un groupe de travail .Généralement utilisée pour attribuer les variables t partagées par les éléments d'un même groupe de travail.
- **Mémoire privée** : Cette région mémoire est privée à un élément de travail. Les variables définies dans la mémoire privée de l'un des éléments de travail ne sont pas accessible aux autres éléments de travail.

Étant donné que la mémoire du programme hôte n'est pas contrôlée par les APIs OpenCL, les mémoire hôte et la mémoire du dispositif OpenCL sont indépendants les unes des autres. Ainsi les interactions entre la mémoire du programme hote et les mémoires de diapositifs se produit dans l'une des deux façons suivantes [38, 69, 78] :

1. en copiant explicitement les données
2. en cartographiant les données dans la mémoire du dispositif OpenCL

3.2.4 Modèle de programmation

Le modèle de programmation OpenCL, et à l'instar de la plupart des langages de programmations modernes, supporte les deux paradigmes majeurs de la programmation parallèle [54, 63] à savoir le parallélisme de données et le parallélisme de tâches.

3.2.4.1 Modèle de programmation parallèlisme de donnés

Ce paradigme définit un calcul en termes de séquences d'instructions appliquée à des éléments multiples d'un objet de mémoire. L'espace d'index associé au modèle d'exécution OpenCL définit les éléments de travail et la façon dont les données seront cartographié (mappées) sur les éléments de travail. Dans un modèle parallèle de données strict, il y a un mappage un-à-un entre l'élément de travail et l'élément dans un objet de mémoire [38, 55, 73, 78].
OpenCL fournit un modèle hiérarchique de programmation de données parallèle proposant aux programmeur deux moyens pour spécifier la répartition hiérarchique :

- **Modèle explicite** : Dans le modèle explicite un programmeur définit le nombre total des éléments de travail à exécuter en parallèle et aussi comment les éléments de travail sont répartis entre les groupes de travail.

- **Modèle implicite :** Dans le modèle implicite, un programmeur ne spécifie que le nombre total des éléments de travail à exécuter en parallèle, alors que la répartition en groupes de travail est gérée par la mise en oeuvre OpenCL [38, 55, 73, 78].

3.2.4.2 Modèle de programmation parallèlisme des tâches

Le modèle de programmation parallélisme des tâches OpenCL définit un modèle dans lequel une seule instance d'un noyau est exécutée indépendamment des autres . Il est logiquement équivalent à l'exécution d'un noyau sur une unité de calcul contenant un seul élément de travail. Selon ce modèle, les utilisateurs expriment le parallélisme en dressant des files d'attentes pour l'ordonnancement des tâches.

3.2.5 Cas d'étude : application addition vectorielle

Les listing suivants 3.2 et 3.1 représentent respectivement une fonction noyau(Kernel) et un programme hôte , tous deux requis pour l'implémentation de l'application la plus simple avec OpenCL "addition de deux vecteurs" , cette implémentation est basée exclusivement sur le parallélisme des données.

Il est simple de faire état sur la complexité programmatique avec OpenCL ,en effet , une simple opération d'addition vectorielle exige aux programmeurs un grand nombre de lignes de code (+ de 30 lignes pour le programme hôte et une dizaine de lignes de codes supplémentaires pour la fonction Kernel) , alors que le C++ AMP par exemple voir le listing 2.2 ,ne nécessite que l'écriture de quelques lignes de codes.

```
float *a_h;
const int N = 2048;
size_t size = N * sizeof(float);
a_h = (float *)malloc(size);
 for (int i=0; i<N; i++) a_h[i] = (float)i;
cl_context context = clCreateContextFromType(0,CL_DEVICE_TYPE_ GPU, NULL, NULL,NULL);
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
 cl_cmd_queue cmd_queue = clCreateCommandQueue(context,devices[0], 0, NULL);
cl_mem memobjs[1];
clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,sizeof(cl_float)*n,a_h,
    NULL);
cl_program program = clCreateProgramWithSource(context,1,count, &program_source,NULL,
    NULL);
cl_int err = clBuildProgram(program,0, ,NULL,NULL,NULL);
 cl_kernel kernel = clCreateKernel(program, square_array, NULL);
err = clSetKernelArg(kernel,0, (void *)&memobjs[0],sizeof(cl_mem));
size_t global_work_size[1] = n;
 clEnqueueNDRangeKernel(cmd_queue, kernel, 1,NULL, global_work_size, NULL,0,NULL, NULL);
 clEnqueueReadBuffer( context, memobjs[0],CL_TRUE,0,n*sizeof(cl_float),a_h,0, NULL, NULL
     );
for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
free(a_h);
```

Listing 3.1 – Programme hôte ; qui additionne deux vecteurs avec OpenCL

```
__kernel void vectorAdd(__global float * a,__global float * b,__global  float * c)
  {
      int nIndex = get_global_id(0);
      c[nIndex] = a[nIndex] + b[nIndex];
  }
```

Listing 3.2 – Programme kernel qui additionne deux vecteurs A et B et stocke le résultat dans le vcteur résultat C

3.3 Politique de transfert des données

De part la nature hétérogène des dispositifs parallèles visés par OpenCL, le groupe Khronos Group a défini une stratégie pour le transfert des données adaptée à chaque catégorie de dispositifs (CPU, APU ou GPU), que nous pouvons répartir en trois politiques orthogonales [16, 94] :

3.3.1 Politique par copie

Il s'agit d'une politique par défaut appliquée aux périphériques de type GPU et implique une opération de copie de données qui utilise le bus PCI Express pour transférer des données entre la mémoire hôte et la mémoire de l'appareil. La spécification OpenCL ne définit pas de processus spécifiques pour le transfert de données. Cependant, à des fins d'illustration, nous présentons une méthode adoptée par AMD dans leur SDK [17] pour transférer des données en utilisant deux opérations (clEnqueueWriteBuffer et clEnqueueReadBuffer). L'exemple nécessite l'exécution du temps d'exécution AMD SDK OpenCL pour effectuer les étapes suivantes :

1. S'assurer que la mémoire est allouée sur la mémoire du GPU.
2. Créer un tampon de mémoire pré épinglé de quelques MO (Protégé des défauts de pages)
3. copie une portion des données source vers des pages de mémoire pré protégées.
4. Épingler les pages de destination dans le système de mémoire de GPU.
5. Effectuer le transfert DMA de la mémoire pré épinglé du CPU vers la mémoire du GPU.
6. Répéter les étapes (3,4 et 5) jusqu'à la fin des transferts

3.3.2 PowerCopy

Les mesures décrites, ont pour effet de ralentir le transfert des données, pour pallier cette lenteur, OpenCL propose deux flags (CL_ALLOC_HOST_PTR, CL_MEM_USE_HOST_PTR), dont la spécification est volontairement ambiguë [38], cette ambigüité est introduite pour permettre aux différents constructeurs d'opérer à des optimisations spécifiques à leurs matériel. À titre d'illustration, le flag "CL_ALLOC_HOST_PTR" est interprété par le runtime de NVIDEA et AMD comme une opération d'allocation d'une zone mémoire épinglée [16, 73], qui permet en théorie les optimisations suivantes :

1. atteindre des débits crêtes pour le transfert des données.
2. éviter un transfert inutile entre la mémoire Host et la mémoire pré épinglée.

3.3.3 Zero-Copy

Il s'agit , d'une politique adaptée aux dispositifs de Type CPU ou APU, où l'implication du bus PCI-Express n'est plus nécessaire pour effectuer des transferts entre la mémoire hôte et la mémoire des dispositifs ; elle désigne par conséquent , une zone mémoire (buffer OpenCL) accessible au programmes hote , qui peut être mise à disposition (cartographiée) des dispositifs parallèles moyennant des appels des deux APIs clEnqueueMapBuffer et clEnqueueUnmapMemObject [38, 55].

3.4 Sémantique des dépendances OpenCL

OpenCL propose deux techniques pour exprimer les sémantiques des dépendances, une première technique dite à gros grains, qui permet d'exprimer des dépendances entre taches [69, 77], à l'opposée de la seconde stratégie, dite à faible grains, qui permet d'exprimer les dépendances propres à chaque tâche (intrinsèque aux fonctions noyau).

En effet, la cartographie (le mapage) des tâches OpenCL représentée par la figure 3.1 est au cœur des sémantiques de dépendances à faible grains [69].
A l'exécution, le runtime OpenCL , met en correspondance le model d'exécution avec le model matériel , cette mise en correspondance est effectuée selon une configuration statique dite Ndrange qui peut être à une , à deux ou à trois dimensions.
le modèle matériel étant une vue abstraite propre à chaque dispositif [69], qui représente un modèle hiérarchique, très proche des architectures des processeurs vectoriels GPU, cette l'hiérarchie matérielle est structurée en deux niveaux :

- le premier niveau défini un ensemble d'unités de traitements,
- le second niveau détermine la capacité de traitements de chaque unité de traitement,

OpenCL introduit le concept des work-items (WIs) qui sont des éléments de travail (des threads), où chaque WI est une instance du noyau dans un espace d'exécution plus large ; cet espace est conçu pour contenir des centaines de milliers de WI, et qui peut être également à une , à deux ou à trois dimensions désigné par Work-Group (WG)

OpenCL pose pour hypothèse que chaque WI sera exécuté par un processeur élémentaire(PE) et chaque unité de traitement est constituée d'un sous-ensemble de processeurs élémentaires (PEs). l'exécution, de Chaque WG sera naturellement cartographiée sur ses PEs.
Cependant les mises en oeuvres OpenCL, limitent la communication pour améliorer l'évolutivité (seuls les threads exécutés par des PEs de la même unité de traitement peuvent se partager les données locales). En d'autres termes, seuls les Work ITEMS associés au même workgroups ont la capacité de se partager leur mémoire locale, ou globale. [38].

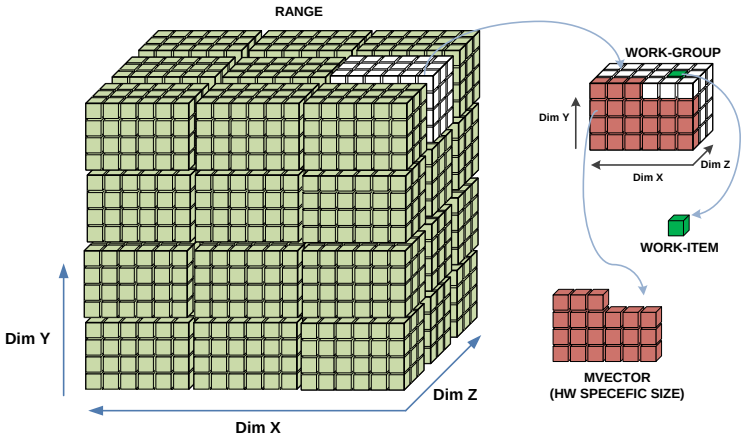


Figure 3.1 – Mappage des données OpenCL

3.5 Évolution des standards OpenCL

3.5.1 OpenCL 1.0

La première version d'OpenCL 1.0 [73] à été publiée par Apple en août 2009 dans l'esprit de proposer une alternative ouverte à la plateforme CUDA de NVIDEA.

3.5.2 OpenCL 1.1

OpenCL 1.1 a été ratifié par le groupe Khronos le 14 juin 2010 [67, 83], en ajoutant des fonctionnalités importantes dans l'esprit d'apporter davantage de flexibilité, de fonctionnalité et aussi et surtout pour améliorer les performances, les apports majeurs de la version 1.1 par rapport à la version 1.0 se résument en ce qui suit :

1. Proposition de nouveaux types de données, essentiellement les vecteurs à 3 composants et le support de formats d'image supplémentaires.

2. Possibilité d'exécuter des traitements sur des objets buffeurs rectangulaires multidimentionnels à une, à deux ou à trois dimensions.

3. Amélioration de l'utilisation des événements pour piloter et contrôler l'exécution des kernels OpenCL.

4. Possibilité d'exécuter des kernels selon les deux modes standards synchrone et asynchrone ; Amélioration de l'interopérabilité OpenGL grâce au partage efficace des images et des objets buffeurs en liant les événements OpenCL à OpenGL.

3.5.3 OpenCL 1.2

Le 15 novembre 2011, le Groupe Khronos a publié la spécification OpenCL 1.2 [38, 55, 68], ce nouveau standard est à ce jour le plus largement utilisé et le plus repandu auprès des programmeurs OpenCL. Il a ajouté des fonctionnalités significatives par rapport aux versions précédentes en termes de performances et de fonctionnalités pour la programmation parallèle. les apports majeurs de la version 1.1 par rapport a la version 1.1 se résument en ce qui suit :

1. Le partitionnement du périphérique : permettant aux programmeurs de réduire la latence pour les tâches critiques en réservant des zones mémoires (des partitions objets subbuffer) dans les dispositifs OpenCL.

2. Le support du moteur DirectX : Le support du DX9 et du DX11 permettant un partage efficace entre OpenCL et DX9 ou DX11.

3. La prise en charge optionnelle du standard IEEE 754 : l'inconvenant majeur des versions 1.0 et 1.1 était le support limité des opérandes à virgule flottante. En effet, les dispositifs GPU sont conçus essentiellement pour le traitement d'image , il n y avait donc aucune nécessité de les doter de technologies avancées pour la prise en charge des calculs mathématique à virgule flottante simple ou double précision pourtant essentiels aux calculs scientifiques. Ainsi, la version 1.2 permet aux programmeurs

d'intéroger compilateurs OpenCL quand a leurs capacités de se conformer aux exigences conformes au standard IEEE 754, si l'implémentation du pilote OpenCL ne les prend pas en charge, les opérandes seront arrondies comme précisé par la spécification IEEE 754 [2].

3.5.4 OpenCL 2.0

Le 18 novembre 2013, le groupe Khronos a annoncé la ratification et la diffusion publique des spécifications OpenCL 2.0 [70] qui apporte pour son essentiel par rapport à la version 1.2 le support :

1. de la Mémoire virtuelle partagée
2. du parallélisme imbriqué
3. de l'espace d'adresse générique
4. la prise en charge des variables atomiques du standard C++11
5. prise en charge d'OpenCL 2.0 pour les systèmes Android

3.5.5 OpenCL 2.1

Le 16 novembre 2015 le groupe Khronos à publié les spécifications provisoires du nouveau standard OpenCL 2.1 [4, 13, 56]. Ce nouveau standard est proposé dans l'esprit de remplacer les kernels OpenCL par des kernels OpenCL du sous-ensemble C++ [4]. Vulkan [8] et OpenCL 2.1 partagent SPIR-V [47] comme une représentation intermédiaire pour la mutualisation des deux compilateurs Vulkan [1] [8] et OpenCL 2.1 [4, 13, 56].

[1] Vulkan d'abord annoncé sous l'appellation OpenGL Next est une interface de programmation graphique proposé par le consortium Khronos Group dans l'optique de remplacer à terme OpenGL

3.5.6 OpenCL 2.2

Le 16 mai 2017 le groupe Khronos à publié les spécifications provisoires pour la ratification OpenCL 2.2 , cette nouvelle ratification porte essentiellement sur [11, 14, 20, 23] :

1. la prise en charge plus large des spécifications C++14 comprenant entre autre les classes objets , les template, les expressions lambda, des surcharges de fonctions et de nombreuses autres constructions pour la programmation générique et la méta-programmation.

2. la nouvelle version préconise également l'usage intégré du nouveau langage intermédiaire Khronos SPIR-V 1.1 [47] qui supporte pleinement les noyaux OpenCL C ++14.

3. les fonctions de bibliothèque OpenCL qui profiteront pleinement de l'adoption du standard C ++14 pour assurer la sureté des programmes tout en accédant à des fonctionnalités telles que les atomiques, les itérateurs, les images, , les pipes et les espaces d'adressage génériques.

4. le stockage des pipes particulièrement utile pour les implémentations FPGA en rendant la taille et le type de connectivité connus au moment de la compilation, dans l'esprit d'améliorer la communication entre les noyaux et les périphériques.

5. le fonctionnement sur n'importe quel matériel compatible avec OpenCL 2.0 (seule la mise à jour du pilote est requise)

3.6 Projets connexes

3.6.1 OpenCL warpper C++

Ratifié en juin 2010 , OpenCl C ++ Wrapper API [21, 36, 37] est une librairie métapogrammé compacte proposée en un seul fichier d'en-tête C ++ cl.hpp qui englobe à la fois les définitions et les implémentations. Elle est conçue dans l'esprit de simplifier le modèle programmatique d'OpenCL par le biais d'API OpenCL C++ qui enveloppent les API primitives de l'OpenCL C 1.1.
L'héritage de classes est autorisé et peut être exploité par les programmeurs pour implémenter des DESL d'un plus haut niveau.

3.6.2 SPIR-V

SPIR-V pour Standard Portable Intermediate Representation [47] son rôle est de compiler les noyaux OpenCL dans un langage intermédiaire interprétable par les dispositifs parallèles. SPIR-V a été initialement développé pour OpenCL et se basait sur LLVM. Depuis sa version 1.0 Khronos le présente comme premier langage ouvert inter-API (OpenGL et OpenCL) pour la représentation native des traitements parallèles (calcul et traitements graphiques) dans un format intermédiaire et interprétable à la fois par les nouvelles ratifications OpenCL(2.1 et 2.2) et par Vulkan. Il permet également :

1. d'optimiser les temps de chargement des noyaux,
2. de proposer un langage front-end commun pour les traitements graphiques et les calculs,
3. d'améliorer la fiabilité et la portabilité des noyaux sur plusieurs implémentations matérielles, et

4. pour les deux nouvelles ratifications OpenCL 2.2 et SPIR-V 1.1 publiées dans le même temps , il est prévu que SPIR-V prenne en charge toutes les phases de compilation intermédiaire des nouveeaux noyaux OpenCL C++

3.6.3 SYCL TM

L'apport le plus significatif de SYCL est qu'il propose un modèle programmatique à l'instar du C++ AMP en une seule source, évitant aux programmeurs la tache ardue et délicate d'écriture et la maintenance de deux programmes séparés (hôte et kernel). SYCL [43, 80] est une couche d'abstraction C ++ multi-plateforme qui s'appuie sur les implémentations d'OpenCL 2.0, cette abstraction est prévue pour améliorer la portabilité et les performances des APIs OpenCL 2.0, tout en apportant davantage de facilité et de flexibilité aux programmeurs OpenCL.

3.7 Conclusion

Contrairement au modèle de programmation parallèle traditionnel à mémoire partagée (OpenMP ,MPI,Pthread. . . etc), le modèle de programmation présenté par OpenCL exige l'écriture de deux programmes séparés. Un programme séquentiel (programme host) et un programme parallèle (programme noyau), ce découplage parallèle-séquentiel permet une approche parallèle formelle stricte mais introduit une complexité supplémentaire. De plus l'absence de librairies orientées objet pour l'écosystème OpenCL limite l'interopérabilité entre OpenCL et la plupart des librairies C++ parallèles ou séquentielles.
Pour simplifier l'utilisation d'OpenCL, plusieurs concepteurs, conscients du potentiel réel pour la production d'applications parallèles hétérogènes, proposent des DESL *(*Domain Embedded Specific Language) métaprogrammés afin de simplifier l'usage des APIs OpenCL et|ou de les optimiser. Le chapitre 4 est consacré à l'étude des techniques usuelles de la métaprogrammation ainsi que leurs applications les plus communes.

CHAPITRE 4

LA MÉTAPROGRAMMATION

4.1 Introduction

Un des sujets de la programmation moderne qui suscite un intérêt de plus en plus croisant auprès des chercheurs , des experts et des concepteurs de librairies, est la métaprogrammation. Bien qu'elle trouve son intérêt avec les generiques des langages Java et C#, elle prend réellement son sens avec les templates de C++ [61].

Le paradigme de la métaprogrammation à été poussé par de nombreux experts et chercheurs de renom, qui ont démontré son intérêt en proposant des librairies aux performances insoupçonnées, en particulier Andrei Alexandrescu, concepteur de la bibliothèque Loki [1] [40], Todd Veldhuizen, concepteur de la bibliothèque Blitz++ [1, 40], Jaakko Järvi, John Maddock, David Abrahams et Aleksey Gurtovoy ; concepteurs de la bibliothèque MPL et tous les trois faisant partie de l'équipe de concepteurs boost [60].

En C ++, les templates ont été initialement introduits au langage C++ pour remplacer l'usage des macros qui échappaient complètement au contrôle et à la rigueur des compilateurs C++ [85], cependant le concept original de la métaprogrammation à l'aide de templates en C++ est en général attribué à Erwin Unruh, qui a proposé en 1994 un programme qui avait la particularité de ne pas compiler sans erreur mais de générer dans ses messages d'erreurs, la liste des nombres premiers jusqu'à un certain seuil , le constat d'Unruh a pour un effet , un réel raz-de-marée et a démontré que les templates en C++ sont un langage complet au sens de la machine de Turing. L'implication est du moins forte en sens et a le mérite de formuler une nouvelle hypothèse programmatique « tout programme susceptible d'être exprimé en C++ à l'exécution l'est aussi, théoriquement, dans son métalangage et

[1] http://loki-lib.sourceforge.net/

donc d'être évalué à la compilation » [10, 12, 40].
Inspiré par la trouvaille d' Unruh ,Todd Veldhuizen, s'est penché sur l'exploitation réelle du potentiel de la métaprogrammation et a proposé plusieurs techniques pour donner vie à une discipline de programmation à part entière, le tout dans l'optique d'écrire une bibliothèque en C++ (Blitz++) capable de battre en terme de performance des programmes Fortran optimisés pour de calculs numériques. Plus formellement, La métaprogrammation est un domaine de programmation où les données sont des programmes. En effet, il s'agit d'une discipline qui permet aux concepteurs de logiciels de spécifier un méta-langage avec un haut niveau d'abstraction décrivant des méta-programmes qui génèrent des programmes de plus bas niveau interprétables par les compilateurs.

La métaprogrammation permet également de définir des familles de classes ou de fonctions paramétrées par abstraction aux types qui sont spécifiés lors des utilisations spécialisées où ces types abstraits sont définis concrètement. Il devient donc possible à travers la méta-programmation de découpler formellement les algorithmes des leurs détails d'implémentation, cette technique qui fait abstraction aux types est plus communément appelée la programmation générique [10, 12, 33, 85]
Dans ce chapitre, nous présentons les concepts de base de la métaprogrammation, ses applications les plus courantes, la STL qui est une librairie standard métaprogrammée qui a fortement contribué au succès du langage et des compilateurs C++, ainsi que l'approche standard pour représenter les conteneurs de données multidimensionnels.

4.2 LES TEMPLATES

Les templates (ou patrons) sont la base de la métaprogrammation en C++. Il s'agit de modèles génériques de code qui permettent de générer automatiquement des fonctions (à partir de patrons fonctions templates) ou des classes (à partir de patron classes templates)

au moment de la compilation. Le template exige donc l'écriture d'un code spécialisé pour un type donné à partir d'un modèle générique. Ces modèles manipulent généralement un type abstrait qui est remplacé par le vrai type C++ au moment de la spécialisation.
Ce type abstrait est fourni au moyen de paramètre template qui peut être un type C++, une valeur (entier, enum, pointeur...) ou même un autre template. La spécialisation d'un template est transparente et invisible et est effectuée lors de la compilation, de manière interne au compilateur et en fonction des arguments donnés au template [88].

4.2.1 Les fonctions génériques

La programmation générique consiste à écrire un code qui n'est pas spécifique à un type en particulier, mais qui peut s'adapter à plusieurs types. À titre d'exemple, pour rendre générique une fonction "somme" (voir le listing 4.1), il suffit de s'affranchir du type que l'on va remplacer par un type abstrait nommé T [12] :

```
1 template <typename T>
2 T  somme (T a, T b)
3  {
4     T resultat = a + b ;
5     return resultat;
6  }
```

Listing 4.1 – Exemple :Fonction générique

1. Le mot clé template indique que la fonction qui suit est une fonction template.

2. Le mot clé typename dans ce contexte sert à déclarer un nouveau type paramétré pour notre nouvelle fonction template. Il est aussi possible d'utiliser le mot-clé class à la place de typename pour la déclaration des paramètres du template.

4.2.1.1 Surdéfinition de Patron et spécialisation

A l'instar des surcharges de fonctions et comme présenté par le linsting 4.2, il est aussi possible de définir plusieurs fonctions templates d'un même nom, possédant des types de paramètres différents ou des expressions différentes. La seule règle à respecter dans ce cas est que l'appel d'une fonction de ce nom ne doit pas conduire à une ambigüité : Un seul Template doit pouvoir être utilisé chaque fois [88].

```
1 template <typename T> T somme (T a, T b) {.....} // patron de fonction
2
3 Char * somme(char *ch1,char *ch2) {.....} // version  pour le type char
```

Listing 4.2 – Exemple :Surdéfinition de Patron

4.2.2 Les Classes génériques

La notion de template permet de définir ce qu'on nomme aussi les « classes génériques », et ce, à l'aide d'une seule définition comportant des paramètres expression, on décrit toute une famille de classes ; le compilateur fabrique (instancie) la ou les classes nécessaires à la demande (on nomme souvent ces instances des « classes templates » ou « classes Patron ») [85, 88].

4.2.2.1 Définition d'une classe Template

Comme illustré par le 4.3 , les paramètres de type sont exprimés en les faisant précéder du mot clé class (ou typename) en mentionnant leur type dans une liste des paramètres introduite par le mot template (comme pour les patrons de fonction avec la différence que tous les paramètres – types ou expressions.

```
template<typename T, typename U, typename V, int A> class MaClasse {};
```

Listing 4.3 – Exemple :Définition d'une classe template

4.2.2.2 Les attributs d'une Classe Template

Comme illustré par le listing 4.4, le type générique étant reconnu par le compilateur à l'intérieur de la classe, il est donc utilisable pour instancier des attributs.

```
template<typename T,typename U> class MaClasse
{
private:
   T attributs1 ;    // attribut de type T
   U attributs2 ;    // attribut de type U
};
```

Listing 4.4 – Example : Attributs d'une Classe template

4.2.2.3 Le Constructeur

Il est aussi possible de spécifier (voir le listing 4.5) au besoin des constructeurs génériques , pour ce faire , on utilise simplement le type T défini avant la classe [10, 85] :

```
template <typename T>
class MaClasse
{
 public:
     MaClasse(T parametre_1)  : attibut1(parametre_1) {}
};
```

Listing 4.5 – Exemple :les constructeurs générique

4.3 Spécialisation des templates

Lorsqu'une fonction ou une classe Template a été définie, il est possible de la spécialiser pour un certain jeu de paramètres Template (pour un ou plusieurs type spécifique).

4.3.1 Spécialisation d'une fonction Template :

Comme illustré par le listing 4.6, pour specialiser une fonction template (préciser une implémentation pour un jeu de paramètre spécifique), seule la spécialisation totale est permise , il est donc de vigueur de signaler au compilateur la définition complète de la fonction par la ligne suivante : template <> L'exemple suivant est une illustration de la spécialisation totale d'une fonction factorielle [85, 88].

```
1  template <unsigned int N>
2  struct Factorielle
3  {
4     enum { valeur = N * Factorielle<N - 1>::valeur };
5  };
6  template <>
7  struct Factorielle<0>
8  {
9     enum { valeur = 1 };
10 };
```

Listing 4.6 – Exemple :Spécialisation d'une fonction Template

4.3.2 Spécialisation des classes templates

Contrairement à la spécialisation des fonctions templates , la spécialisation des classes templates peut être totale ou partielle. Les spécialisations partielles permettent de définir l'implémentation d'une fonction ou d'une classe Template pour certaines valeurs de leurs paramètres Template, et de garder d'autres paramètres indéfinis. Il est même possible de

changer la nature d'un paramètre Template (c'est-à-dire préciser s'il s'agit d'un pointeur ou non) et de forcer le compilateur à prendre une implémentation plutôt qu'une autre, selon que la valeur utilisée pour ce paramètre est elle-même un pointeur ou non.

```
// Definition dune classe template :
template <class T1, class T2, int I> class A {};
// Specialisation partielle n_1 de la classe :
template <class T, int I> class A<T, T*, I> {};
// Specialisation partielle n_2 de la classe :
template <class T1, class T2, int I> class A<T1*, T2, I>
{};
// Specialisation totale
template <> class A<int, double *, 5> {};
```

Listing 4.7 – Exemple :Spécalisation partielle et totale d'une classe template

4.3.3 Les métafonctions

Il arrive souvent qu'un programmeur ait le besoin d'effectuer des traitements sur des constantes connues à la compilation, l'idée derrière les fonctions template est justement d'effectuer ces traitements par le compilateur (au moment de la compilation et de décharger ainsi complètement le programme compilé de ces calculs), un fait, qui devrait en principe apporter quelques optimisations aux programmes générés . Pour mieux illustrer ce principe, nous prenons l'exemple présenté par le listing 4.6 d'une métafonction qui calcule le factoriel d'un entier au compile time [40] :

4.4 Les traits et les politiques

4.4.1 Les traits

Les traits sont une technique de métaprogrammation qui permet de prendre des décisions en temps de compilation en fonction des types, tout comme dans le génie logiciel ou il

est souvent nécessaire de prendre des décisions d'exécution basées sur des valeurs. En ajoutant un niveau supplémentaire d'indirection, les traits permettent donc de prendre les décisions liées à un type de décision en dehors du contexte immédiat dans lequel elles sont faites. Cela rend le code plus lisible et plus facile à entretenir [10, 40].

4.4.2 Les politiques

Les politiques et les classes de politiques aident à mettre en œuvre des éléments de conception orthogonaux, efficaces et hautement personnalisables. Une stratégie ou politique définit une interface de classe ou une interface de modèle de classe.
Les politiques ont beaucoup de points communs avec les traits, mais diffèrent en ce sens qu'elles accordent moins d'importance aux types et plus d'accent sur le comportement [84, 88].

4.5 LES EXPRESSIONS TEMPLATES

Les modèles d'expression ont été jugés particulièrement utiles par les auteurs de bibliothèques pour l'algèbre linéaire. Parmi les bibliothèques qui utilisent le modèle d'expression, on trouve une large variété de librairies [29] parmis lesquelles VexCL [90] , uBLAS, Eigen et Blitz ++ [89].

Les modèles d'expression sont une technique de métaprogrammation par template en C ++ qui crée des structures représentant un calcul au moment de la compilation, ces structures seront évaluées uniquement au besoin pour produire un code efficace pour l'ensemble du calcul. Il est donc possible aux programmeurs C++ d'obtenir des optimisations en contournant l'ordre naturel de l'évaluation du compilateur, la fusion de boucles en est un exemple d'optimisation permis par ces techniques [12, 85].

Les modèles d'expression ont été inventés de manière indépendante par Todd Veldhuizen et David Vandevoorde et c'est Veldhuizen qui a donné le nom de modèle d'expression (Expression template) à cette technique et c'est devenu des techniques populaires pour la mise en œuvre du logiciel d'algèbre linéaire [12].

4.5.1 L'arbre syntaxique

Les expressions templates sont utilisés pour différer l'évaluation des calculs, ce qui revient à la génération d'un code optimisé dans la définition surchargé de l'opérateur (=) [88]. En effet, au lieu de calculer des résultats intermédiaires à l'aide de classes templates, l'opérateur = va ensuite parcourir cet arbre syntaxique et effectuer le calcul en une seule et unique opération [12].

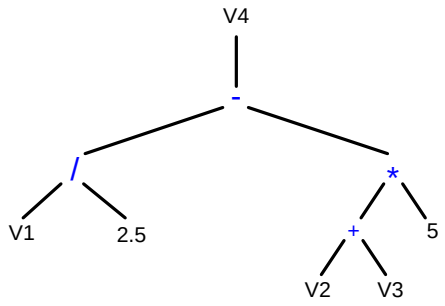


Figure 4.1 – Exemple d'arbre syntaxique

l'arbre syntaxique créé ne sera donc pas physique, il sera construit et parcouru par le compilateur. Une fois le programme compilé, il n'en restera que la ligne de calcul donnée plus haut.

4.5.2 Exemple d'utilisation

Considérons une bibliothèque représentant des vecteurs et des opérations qui s'appliquent sur ces mêmes vecteurs. Une opération mathématique commune consiste à additionner deux vecteurs u et v , pour produire un nouveau vecteur résultat. L'implémentation naïve en C ++ de cette opération serait une classe vecteur Vec avec une méthode surchargée de l'opérateur + qui renvoie un nouvel objet vecteur.

Les utilisateurs de cette classe utiliseront cette classe Vec x = a + b ; où a et b sont les deux instances de Vec.

Le problème avec cette approche est que des expressions plus compliquées telles que Vec x = a + b + c sont mises en œuvre de manière inefficace. La mise en œuvre produit d'abord un vecteur temporaire pour contenir a + b, puis produit un autre vecteur avec les éléments de c ajoutés. Même avec l'optimisation de la valeur de retour, le compilateur aurra tendance à allouer de la mémoire temporaire au moins deux fois et le compilateur aura tendence à générer deux boucles.

L'évaluation retardée résout ce problème et peut être implémentée en C ++ en laissant l'opérateur + renvoyer un objet d'un type personnalisé, VecSum, qui représente la somme non évaluée de deux vecteurs, ou un vecteur avec un VecSum. Les modèles d'expression implémentent une évaluation différée à l'aide d'arbres d'expression qui n'existent qu'au moment de la compilation. Chaque affectation à un Vec, tel que Vec x = a + b + c, génère un nouveau constructeur Vec si nécessaire par instanciation du modèle. Ce constructeur fonctionne sur trois Vec ; il alloue la mémoire nécessaire, puis exécute le calcul. Ainsi, une seule affectation de mémoire est effectuée.

Un exemple de mise en œuvre des modèles d'expression est présenté par les linstings 4.8, 4.9 et 4.10. Une classe de base VecExpression représente une expression vectorielle, Cet exemple est basé sur l'expression réelle type E à mettre en œuvre, selon le modèle récurrent.

La classe Vec stocke toujours les coordonnées d'une expression vectorielle entièrement évaluée et devient une sous-classe de VecExpression.La somme de deux vecteurs est représentée par un nouveau type, VecSum, qui est modélisé sur les types de l'expression gauche et droite de la somme afin qu'il puisse être appliqué à des paires de vecteur d'expressions vectorielles. Un opérateur surchargé + permet l'expression syntaxique du constructeur VecSum.

```
template <typename E>
 class VecExpression {
  public:
  double operator[](size_t i) const { return static_cast<E const&>(*this)[i];      }
 size_t size()               const { return static_cast<E const&>(*this).size(); }
  operator E& () { return static_cast<E&>(*this); }
 operator const E& () const { return static_cast<const E&>(*this); }
 };
```

Listing 4.8 – Prototype de mise en oeuvre de VecExpression

.

```
class Vec : public VecExpression<Vec> {
    std::vector<double> elems;
  public:
    double operator[](size_t i) const { return elems[i]; }
    double &operator[](size_t i)       { return elems[i]; }
    size_t size() const                { return elems.size(); }
    Vec(size_t n) : elems(n) {}
    template <typename E>
    Vec(VecExpression<E> const& vec) : elems(vec.size()) {
        for (size_t i = 0; i != vec.size(); ++i) {
            elems[i] = vec[i];
        }
    }
};
```

Listing 4.9 – Exemple :Mise en oeuvre de la classe vect héritée de la calasse de base VecExpression

Avec les définitions ci-dessus, l'expression a + b + c est de type VecSum<VecSum<Vec, Vec>, Vec> , ainsi Vec x = a + b + c invoque le constructeur Vec du modèle avec l'opération vcetsum comme argument du modèle de l'expression E.

```
template <typename E1, typename E2>
class VecSum : public VecExpression<VecSum<E1, E2> > {
  E1 const& _u;  E2 const& _v;
  public:
    VecSum(E1 const& u, E2 const& v) : _u(u), _v(v) {}
   double operator[](size_t i) const { return _u[i] + _v[i]; }
   size_t size()               const { return _v.size(); }};
template <typename E1, typename E2>  VecSum<E1,E2> const
 operator+(E1 const& u, E2 const& v) {return VecSum<E1, E2>(u, v);}
```

Listing 4.10 – Exemple :Opération syntaxique Vecsum

À l'intérieur de ce constructeur, le corps de la boucle elems[i] = v[i] est effectivement développé (suite aux définitions récursives de operateur + et operateur [] sur ce type) à elems[i] = a.elems[i] + b.elems[i] + c.elems[i] ; sans vecteurs temporaires nécessaires avec un seul parcour de chaque bloc de mémoire.

4.6 Les pointeurs intelligents

Les Pointeurs intelligents C/C++ ou (smart pointer) sont des classes métaprogrammées qui enveloppent des pointeurs nus, tout en offrant une sémantique d'un plus haut niveau. C'est principalement autour des opérations liées à la durée de vie des pointeurs (création, copie, destruction...) que les pointeurs intelligents apportent de la valeur aux programmeurs [53]. Cette notion de pointeur intelligent à été introduite au standard, qui a présenté l'auto_ptr , bien que cette première tentative a apporté davantage de problèmes que de solutions relative à la gestion de la durée de vie des objets enveloppés par auto_ptr , néanmoins, conscient du potentiel des pointeurs intelligents pour faciliter la gestion de la mémoire

dynamique ainsi que la gestion du cycle de vie des objets gérés par les pointeurs intelligents, les standards C++0x , C++11 , C++14 et C++17 ont continué à proposer des jeux de pointeurs [85] (std : :shared_ptr, std : :weak_ptr et std : :unique_ptr) comme alternative stable au std : :auto_ptr.

- unique_ptr qui est destiné à pointer un objet de manière unique (relation de 1 à 1).
- shared_ptr qui est destiné à partager un objet entre plusieurs shared_ptr qui seront en mesure de pointer un même objet, qui partagent un compteur de référence.
- weak_ptr est un pointer intelligent qui n'agit pas sur l'objet pointé : n'agit pas sur le compteur de référence et ne de détruit pas l'objet pointé.

4.7 La Standard template librairie

Le STL est une bibliothèque métaprogrammée qui fournit des solutions à la gestion des collections de données avec des algorithmes génériques [28, 48, 58]. Elle permet aux programmeurs de bénéficier de structures de données et des algorithmes qui font référence dans le paysage C++ moderne. Tous les composants du STL sont des templates, de sorte qu'ils peuvent être utilisés pour des types d'éléments arbitraires.

4.7.1 Les composants de la STL

Le STL est basé sur la coopération de divers composants structurés autour des concepts de découplage des conteneurs, des itérateurs et des algorithmes [28] :

4.7.1.1 Les itérateurs

Les itérateurs sont utilisés pour parcourir les éléments de collections d'objets. Ces collections peuvent être des conteneurs ou des sous-ensembles de conteneurs.

L'interface pour les itérateurs est presque la même que pour les pointeurs ordinaires. Pour incrémenter un itérateur, on fait appel à l'opérateur ++. Pour accéder à la valeur d'un itérateur, on utilise l'opérateur *.
Ainsi, on peut considérer un itérateur à l'instar des pointeurs inteligents comme une abstraction au pointeurs qui ont la capacité de parcourir une structure de données par des déplacements (élément suivant , élément précedent , dernier élément et premier élément" [28, 53].

4.7.1.2 Les Algorithmes

Les algorithmes agissent sur les conteneurs. Ils fournissent les moyens par lesquels les opérations d'initialisation, de tri, de recherche et de transformation du contenu des conteneurs sont effectués. Le concept de STL repose sur une séparation des données et des opérations. Les données sont gérées par des classes conteneur, et les opérations sont définies par des algorithmes configurables. Les itérateurs sont le lien entre ces deux composants (conteneurs et algorithmes). Ils autorisent ainsi à n'importe quel algorithme d'interagir avec tous les conteneurs de la STL [53].

4.7.2 Les conteneurs standards

Les conteneurs sont des objets support qui stockent une collection d'autres objets (leurs éléments). Ils sont implémentés en tant que modèles génériques, ce qui permet une grande flexibilité dans les types pris en charge [48].
Le conteneur gère l'espace de stockage pour ses éléments et fournit des fonctions membres pour y accéder, soit directement, soit via des itérateurs . Pour répondre aux différents besoins liés aux mises en oeuvres d'algorithmes réguliers et irréguliers , la STL fournit différents types de conteneurs, parmi lesquels on trouve :

4.7.2.1 Les conteneurs de séquence

Les conteneurs de séquence sont des collections ordonnées dans lesquelles chaque élément a une certaine position. Cette position dépend du moment et du lieu de l'insertion, mais elle est indépendante de la valeur de l'élément. Par exemple, si on place six éléments dans une collection ordonnée en ajoutant chaque élément à la fin de la collection, ces éléments sont dans l'ordre exact où on vient de les placer. La STL contient cinq classes de conteneurs de séquences prédéfinies : vector, array , deque, list et forward_list.

- **std : :vector** est un conteneur séquentiel qui représente les vecteurs à une dimension c++ de taille dynamique ,la taille du vecteur peut être augmentée ou diminuée automatiquement. Ses éléments sont stockés de façon contigüe, et accessibles non seulement via les itérateurs, mais aussi à partir des pointeurs C. Les vecteurs (std : :vector) occupent généralement plus d'espace que les tableaux statiques, du fait que de la mémoire supplémentaire est allouée pour anticiper un accroissement future. Ainsi, un vecteur std : :vector n'a pas besoin de ré-allouer la mémoire chaque fois qu'un élément est inséré, mais seulement lorsque la mémoire additionnelle est épuisée.
  La complexité (efficacité) des opérations courantes sur les vector sont les suivantes :

  1. Accès aléatoire - constante O(1)
  2. Insertion ou suppression d'éléments à la fin - O(1)
  3. Insertion ou retrait d'éléments - linéaire O(n)

- **std : :array** est un conteneur séquentiel qui représente des vecteurs de taille constante. Cette structure possède la même sémantique de type agrégat (il n'a pas de constructeurs et aucun membre en privé ou protégé) qu'un tableau de style C. La taille et l'efficacité de array<T,N> pour un certain nombre d'éléments sont équivalentes à celles du tableau T[N] de style C correspondant.

- **std : :deque** cette structure offre les avantages d'un conteneur standard, et implémente la sémantique des itérateurs à deque (Queue à double entrées), elle est un conteneur de séquence indexées qui permet une insertion et suppression rapide à ses extrémités. Par opposition à std : :vector, les éléments d'une file double ne sont pas stockés de façon contiguë : typiquement chaque élément du tableau est alloué dans un espace mémoire séparé , ainsi il évite les opérations de copies mémoires et les déplacements de zones couteuses en temps et en mémoire lors du redimensionnement du conteneur.

    1. La complexité des opérations communes sur deques est la suivante :
    2. Accès aléatoire O(1) constante
    3. Insertion ou suppression d'éléments à la fin ou au début -O(1)
    4. Insertion ou suppression d'éléments à un emplacement quelconque O(n) linéaire

- **std : :list & std : :forword_list** : sont deux conteneurs séquentiels implémentés pour fournir respectivement une implémentation standard des chaines doublement et simplement liées , ils supportent tous deux l'insertion et la suppression des éléments en temps constant, et à l'instar de deque, chacun des éléments des listes est stocké dans une zone mémoire séparée . Cependant l'accès aléatoire n'est pas pris en charge et seul un parcourt linéaire est supporté.

4.7.2.2 Les conteneurs associatifs

Les conteneurs associatifs sont des collections triées dans lesquelles la position d'un élément dépend de sa valeur (ou clé, si c'est une paire clé / valeur) en raison d'un certain critère de tri. Si on construit une collection associative composée de six éléments, leurs

valeurs déterminent leur ordre. L'ordre d'insertion n'a pas d'importance. Le STL contient quatre classes de conteneurs associatifs prédéfinis : set, multiset, map et multimap.

- **std : :set & std : :multiset** : std : :set est un conteneur associatif qui contient un ensemble trié d'objets de type clé. Tandis que Std : :multisets sont des conteneurs qui stockent des éléments suivant un ordre spécifique et où plusieurs éléments peuvent avoir des valeurs équivalentes,cependant, la valeur des éléments dans un multiset ne peut pas être modifiée une fois dans le conteneur (les éléments sont toujours const), mais ils peuvent être insérés ou retirés du conteneur.
- **std : :map & std : :multimap** std : :map est un conteneur trié associatif contenant les paires clé-valeur avec une clé unique, tandis que le multimap est un conteneur associatif qui stocke également des éléments formés de paires clé-valeur, mais avec la particularité qu'un où plusieurs éléments peuvent avoir une même clé.

4.8 Les conteneurs de données C++ et la dimensionnalité

4.8.1 Les conteneurs standard et la dimensionnalité

Malgré l'importance des tableaux multidimensionnels, jusqu'à présent, les compilateurs C ++ souffrent d'un manque de représentation dimensionnelle [51, 59]. Effectivement et depuis les versions antérieures jusqu'à la dernière version C++17, le C++ a utilisé les tableaux multidimensionnels de données POD, qui sont des conteneurs de données statiques (non redimensionnables) résidant dans la mémoire pile. La norme ISO C99 a préconisé des tableaux à longueur variable (VLA, qui sont des conteneurs de mémoire contigus également résidants dans la mémoire pile. Par opposition aux conteneurs POD, la taille des VLA peut varier pendant l'exécution, bien malencontreusement, en raison de leur faible

capacité de stockage, les VLA ont été rapidement abandonnés en faveur des conteneurs std : :valarray [28, 48, 58] proposés par la norme C ++ 0x. ValArray est un conteneur de données contigu unidimensionnel qui peut imiter le comportement de conteneurs multi-dimensionnels en utilisant des méthodes d'accès par calcul d'indexe. Malheureusement l'expérience valArray s'est également soldée par un puisant échec et n'a pas réussi à atteindre le succès attendu [48].

Plus récemment, Lawrence Crowl et Matt Austern [6] ont proposé une nouvelle classe de conteneurs appelés dynarray en prévision de les inclure dans le nouveau standard c++14, les dynarray sont des conteneurs non redimensionnables [25], conçus pour être simples et compacts favorisant des allocations dans la pile plutôt que dans le tas mémoire. Malheureusement, ce choix, bien que judicieux de favoriser la mémoire pile, s'est avéré mal adapté à la plupart des compilateurs C ++ modernes. Le résultat est que les dynarray ont été retirés du standard c++14 et c++17.

En absence de conteneurs standards multidimensionnels, la communauté des utilisateurs C++ a tendance d'utiliser les conteneurs std : :vector pour représenter les systèmes multi-dimensionnels [85] comme illustré par le linsting 4.11 par la sémantique de vector of vector ... of vector. Cette sémantique bien que correcte, néanmoins, elle produit une grande surcharge d'allocations mémoires et des conteneurs à mémoire disparates.

```
// Create a 3D array that is 3 x 4 x 5
#include <vector>
vector<vector<vector<double>>> A(3, vector<vector<double>>(4, vector<double>(5)));
int values = 0;
  for(index i = 0; i != 3; ++i)
    for(index j = 0; j != 4; ++j)
      for(index k = 0; k != 5; ++k)
        A[i][j][k] = values++;
```

Listing 4.11 – Exemple :Génération d'une matrice à trois dimensions avec la sémantique vecteur de vecteur de vecteur

4.8.2 L'expérience boost : :multiarray

```
#include "boost/multi_array.hpp"
#include <cassert>
int main ()
{
  typedef boost::multi_array<double, 3> array_type;
  typedef array_type::index index;
  array_type A(boost::extents[3][4][5]);
  int values = 0;
  for(index i = 0; i != 3; ++i)
    for(index j = 0; j != 4; ++j)
      for(index k = 0; k != 5; ++k)
        A[i][j][k] = values++;
  return 0;
}
```

Listing 4.12 – Exemple :Génération d'un tableau à trois dimensions avec boost : :multiarray

Comme ilustré par le listing 4.12 , La bibliothèque boost : :multiarray [34, 35, 81] fournit un modèle de classe pour les tableaux multidimensionnels , ainsi que des adaptateurs sémantiquement équivalents pour les tableaux de données contiguës. Les classes de cette bibliothèque implémentent une interface commune, formalisée comme un concept de programmation générique.

La conception de l'interface est aux normes boost et de la STL. boost : :multiArray est un moyen plus efficace et plus pratique d'exprimer les tableaux N-dimensionnels, que les alternatives existantes (la formulation std : :vector <std : : vector <... » des tableaux N-dimensionnels). Les tableaux fournis par la bibliothèque sont accessibles au moyen d'une syntaxe par opérateur subscript standard des tableaux POD de C ++. Des fonctionnalités supplémentaires, telles que le redimensionnement,et la création des vues, sont également proposées par cette librairie.

boost : :multiarray [34] ,qui est à notre connaissance l'unique libraire qui propose une

représentation multidimensionnelle concrète. Cependant , elle produit des conteneurs à mémoire disparates et utilise des hiérarchies objets complexes qui dégradent considérablement ses performances. Elle est par conséquent rarement utilisée dans des applications aussi bien séquentielles que parallèles où la performance est exigée.

4.8.3 Les conteneurs multidimensionnels et l'écosystème OpenCL

OpenCL est incompatible avec la totalité des conteneurs orientés objet [69]. Pour palier le manque des conteneurs de données pour OpenCL, les concepteurs de librairies STL les plus populaires ciblant des exécutions sur GPU Bolt [15], Boost.Comput [60] et Thrust [72] limitent l'utilisation des conteneurs de données à une exploitation à une seule dimension.

Thrust est une bibliothèque développée par NVIDIA, qui cible les périphériques GPU du même constructeur (NVIDIA), elle est basée sur CUDA pour orchestrer les exécutions sur GPU ainsi qu'intel TBB [24] et OpenMP [5] pour les exécutions multi-cœur. Thrust fournit un conteneur vector : :host_vector à une seule dimension représentant des collections de données contiguës allouées à la mémoire hôte et un second conteneur nommé device_vector également à une seule dimension alloué dans la mémoire du GPU.

Bolt est une bibliothèque qui s'inscrit dans la même logique que Thrust, développée par AMD , elle exploite plusieurs backend (OpenCL, C ++ MPL et Intel TBB). Contrairement à Thrust , Bolt ne propose aucun conteneur spécialisé ; elle repose entièrement sur les conteneurs de la STL, plus en particulier std : : vector.

Boost.compute offre le même niveau d'abstraction et des fonctions similaires aux bibliothèques de Thrust et Bolt, mais par opposition à Thrust et Bolt , Boost.compute repose entièrement sur OpenCL. A l'instar de Bolt, Boost.compute utilise les conteneurs unidimensionnel std : : vector de la STL. Elle offre également une abstraction de haut niveau (boost : :computer : : vector) et OpenCL API pour simplifier les opérations liées aux allocations mémoire ou de copie de données sur des périphériques GPU.

4.9 Conclusion

Dans ce présent chapitre , nous avons présenté les concepts de base de la métaprogrammation , ainsi que les librairies les plus réputées basées sur cette technique programmatique. À l'exception de boost : :multiarray, aucune librairie ne propose un contexte formellement défini pour la production et la manipulation des conteneurs multidimensionnels.Cependant , boost : :multiarray n'a pas été conçue dans l'esprit de s'utiliser avec les plateformes parallèles hétérogènes qui exigent un stockage des données contigües.
Pour palier le manque de conteneurs de données multidimensionnels adaptés pour la programmation parallèle hétérogène, nous nous sommes fixé pour objectif la conception et la mise en œuvre d'une nouvelle librairie métaprogrammée performante avec un haut niveau d'abstraction , conforme aux standards c++11 et simplement utilisable avec les nouvelles plateformes destinées à la programmation parallèle hétérogène. Le chapitre 5 présente les aspects conceptuels de notre nouvelle librairie que nous baptisons CL_ARRAY.

CHAPITRE 5

CONCEPTS DE LA BIBLIOTHÈQUE CL_ARRAY

5.1 Introduction

L'avenir du frameworks OpenCL , bien que prometteur , est étroitement lié à sa capacité de proposer un eenvironnement qui conjugue à la fois portabilité, stabilité, simplicité, sécurité et performances, cependant , le ralliement de ces objectifs, ne pourrait se concrétiser , en l'absence de conteneurs de données , évolués conçus spécifiquement pour l'écosystème OpenCL.

Dans le contexte particulier lié aux conceptions des conteneurs de données , beaucoup de framwork's destinés à la programmation parallèle proposent des conteneurs avec des extensions sémantiques pour exprimer le parallélisme de données [24, 63, 79]. En effet, C++ AMP introduisent de nouveaux concepts pour la programmation parallèles pour les dispositifs GPU cite10, par la spécification des techniques d'accès basées sur des type index, le tiling comme technique de partitionnement, et des enveloppes objets array_view appliquées à des collections de données contigües (std : :vector à une dimension , tableau POD à une dimension ou encore std : :array aussi à une dimension), conçues pour permettre les transferts implicites des données entre la mémoire du programme hôte et la mémoire du dispositif parallèle (GPU).

Fort de l'expérience C++AMP , et conscient du potentiel du array_view pour exprimer la dimensionnalité , Łukasz Mendakiewicz et Herb Sutter ont soumis une proposition, pour intégrer array_view, comme vue abstraite multidimensionnelle, bounds et index comme primitive d'accès et enfin bounds_iterator pour l'interopérabilité avec les algorithmes STL, cette proposition a été validée et sera probablement proposée par les nouveaux compilateurs de la nouvelle norme C++17 [59].

Toutefois, array_view du C++AMP, n'est pas utilisable dans l'écosystème OpenCL, néanmoins, l'approche adoptée par les concepteurs du C++AMP basée sur les tableaux unidimensionnels et array_view pour exprimer la dimensionnalité est transposable pour l'écosystème OpenCL.
Ce chapitre présente la conception de notre librairie que nous baptisons CL_ARRAY de conteneurs multidimensionnels réguliers et contigus destinés à l'écosystème OpenCL, CL_ARRAY est compatible avec le standard 1.1 [67] le plus répandu et pris en charge par la majorité des dispositifs OpenCL.
La nouvelle librairie à l'opposé de l'approche suivie par les concepteurs du C++ AMP est constituée d'un ensemble de trois conteneurs de données tous multidimensionnels (HEAP , STACK , et OCL), et une structure OCL_ARRAY_VIEW sémantiquement proche du array_view du C++AMP mais adaptée à OpenCL, que nous proposons comme interfaçage entre les conteneurs du CL_ARRAY et les APIs OpenCL, permettant entre autre de formaliser le modèle d'exécution et|ou le modèle de dépendances à faible grains d'OpenCL aux moyens de transformations multidimentionnels.
Bien que les conteneurs du CL_ARRAY sont conçus pour l'écosystème OpenCL, il n'en demeure pas moins qu'ils sont dotés à travers des enveloppes OCL_ARRAY_VIEW d'itérateurs standards, simplifiant leur utilisation avec la plupart des autres plateformes parallèles populaires (OpenMP [5], MPI [3], C++ AMP [65] et Intel TBB [44]). Cette compatibilité est prévue pour favoriser la cohabitation entre différentes plateformes parallèles et simplifier les processus de migration des programmes séquentiels ou parallèles vers OpenCL.

5.2 Concepts de CL_ARRAY

La philosophie véhiculée par la librairie CL_ARRAY, s'articule autour d'une proposition sémantique alternative à celle présentée par Lawrence Crowl et Matt Austern concepteurs

du dynArray [6], pour représenter les conteneurs multidimensionnels à comportement statique (non-redimensionnables). Les conteneurs CL_ARRAY sont basés sur des expressions syntaxiques métaprogramées plus simples, plus compactes et surtout plus intuitives. En effet, nous proposons trois conteneurs à comportement statique (CL_ARRAY : :HEAP , CL_ARRAY : :OCL et CL_ARRAY : :STACK) tous construits selon une syntaxe , une sémantique et une méthode d'accès unifiée. Contrairement à la proposition de Lawrence Crowl et Matt Austern [6] qui souhaitent concéder la décision aux compilateurs quand à l'emplacement mémoire des conteneurs dynarray (mémoire du tas ou mémoire pile), nous concédons ce choix aux développeurs. En effet , un développeur à travers la syntaxe , la sémantique et la méthode d'accès unifiées proposées pour les conteneurs CL_ARRAY , peut à tout moment modifier l'emplacement mémoire par un simple changement déclaratif sans aucune autre modification dans son code.
Pour ce qui est des conteneurs multidimensionnels qui supportent les opérations de redimensionnement, d'insertions et de suppression des éléments ; nous proposons une syntaxe et une sémantique proche de celles utilisées par boost : :multiarray [35] mais avec de nouveaux algorithmes pour améliorer les performances et surtout pour garantir la contiguïté des données.

CL_ARRAY::STACK

Il s'agit d'un tableau multidimensionnel statique résidant dans la mémoire pile (stack) équivalent à std : :array [1] qui enveloppe des tableaux POD multidimensionnels au lieu d'un tableau uni-dimensionnel POD enveloppé par std : :array..

[1] http://en.cppreference.com/w/cpp/container/array

CL_ARRAY::**HEAP**

Il s'agit d'un tableau multidimensionnel résidant dans la mémoire de tas (heap), que nous proposons avec deux sémantiques adaptées : l'une identique a celle proposée pour les conteneurs non redimensionnables CL_ARRAY::STACK et une autre pour les conteneurs dynamiques redimensionnables proche de celle proposée par boost : :multiarray.

CL_ARRAY::**OCL**

Il s'agit d'un tableau multidimensionnel alloué à la fois dans la mémoire heap et la mémoire du dispositif OpenCL , il utilse la même sémantique et la syntaxe prévue pour CL_ARRAY : :HEAP non-redimentionable et CL_ARRAY : :STACK . Il est conçu pour accélérer le transfert des données entre la mémoire hôte et les périphériques OpenCL.

5.2.1 CL_ARRAY : :STACK

5.2.1.1 Positionnement du problème

Les tableaux multidimensionnels C ++ POD (c.-à-d. Int arr[2][2][2]) sont dépourvus d'itérateurs et par conséqent ils n'interagissent pas correctement avec la bibliothèque standard C ++ . Pour résoudre ce problème, les standards C++11 et supérieur proposent std : :array, un conteneur qui encapsule des tableaux POD à une dimension et implémentent la sémantique complète des iterateurs. std : :array peut être utilisé pour implémenter des tableaux N-dimensionnels par la formulation std : :array de std : :array ... de std : :array, cependant le standard ne garantit pas la contiguité des éléments des tableaux multidimentionnels générés par cette formulation.

5.2.1.2 Notre proposition

Pour ganrantir la contiguité des données, nous proposons CL_ARRAY : :STACK, qui encapsule des tableaux POD multidimensionnels. Cette structure a la même sémantique qu'un tableau multidimensionnel C-style. La taille et l'efficacité d'un tableau $[a_0]$ $[a_1]$ ... $[a_{Nd-1}]$ sont équivalentes à celles du tableau multidimensionnel POD de style C $[a_0]$ $[a_1]$ ... $[a_{Nd-1}]$. La section 5.4 décrit l'interfaçage des conteneurs CL_ARRAY : :STACK avec OCL_ARRAY_VIEW, où un pointeur unidimensionnel sur le tableau POD est requis. À cette fin, nous avons développé un modèle de structure récursive Update1dptr , dont la sémantique est décrite dans le tableau 5.IX.

5.2.2 CL_ARRAY : :HEAP & CL_ARRAY : :OCL

5.2.2.1 Positionnement du problème

Pour autant que nos connaissances nous permettent de l'affirmer, seules boost : : multiarray, et quelques bibliothèques std style avec la sémantique "vecteur par vecteur de vecteur de vecteur ... de vecteur" tels std : :vector , _vector d'intel TBB [2], fournissent des tableaux multidimensionnels dynamiques C ++ avec une méthode d'accès par opérateur subscript C++ style [], ces bibliothèques produisent des conteneurs multidimensionnels à mémoire clairsemée (tableaux disparates) et présentent par conséquent quelques problèmes que nous énumérons comme suit :

1. la construction des tableaux multidimensionnels nécessite un nombre important d'opérations d'allocations et deallocations mémoire exprimées par la formule 5.1.
2. les tableaux multidimensionnels se caractérisent par une très mauvaise localité mémoire spatiale et temporelle ; en général pour un réseau Nd multidimensionnel avec

[2] https://software.intel.com/en-us/node/506203

des dimensions $[a_0][a_1]...[a_{Nd-1}]$, produisent un stockage mémoire segmenté en $Nbr_{Locality}$=$[a_0]$*$[a_1]$*...*$[a_{Nd-2}]$, cette segmentation a pour effets néfastes :

(a) de réduire l'interopérabilité entre les tableaux multidimensionnels et les APIs OPenCL , elle nécessite un vecteur contiguë supplémentaire alloué dans la mémoire de tas et des copies multiples chronophages ($Nbr_{Locality}$) entre la mémoire hôte et l'objet Buffer OpenCL.

(b) d'augmenter les défauts de caches et, par conséquent, implique une forte dégradation des performances du programme hôte et des APIs OpenCL.

Alternativement, pour palier la surcharge des opérations d'allocation et déallocation méomoire et aussi et surtout pour améliorer la performance des tableaux multidimensionnels C ++, de nombreux concepteurs de bibliothèques, à l'instar des concepteurs Blitz ++, optent pour des pseudo-arrays qui imitent des tableaux multidimensionnels par des opérations de calcul d'index. De tels pseudo-array présentent les avantages suivants :

1. Zero surchage mémoire : Contrairement aux tableaux C++ style, les pseudo-arrays ne nécessitent aucune mémoire supplémentaire et par conséquent ils ne requièrent qu'une seule opération d'allocation et de deallocation mémoire.

2. Nativement adapté avec les APIs OpenCL : Avec les pseudo-arrays, les éléments sont stockés dans un ordre contigu et présentent , en effet :

   (a) Les pseudo-arrays ont la capacité de s'utiliser nativement avec les API OpenCL par la définition d'un pointer unidimentionnel sur le premier élément de la collection de données et sont donc adaptés aussi bien pour des stratégies de copie par défaut et que des stratégies de copies Zero-Copy.

   (b) Les pseudo-arrays entraînent un nombre réduit de défauts de caches , ce qui améliore les performances du programme hote et des APIs OpenCL.

Cependant, les pseudo-arrays présentent quelques inconvénients majeurs que nous déclinons comme suit :

1. Une performance dépendante de l'implémentaion :

   Les pseudo-arrays sont souvent utilisés avec une méthode d'accès style Fortran en surchargeant l'opérateur () tel que proposé par Todd Veldhuizen [3], qui a obtenu une implémentation de modèle d'accès direct (sans aucune indirection) , cette optimisation est obtenue au dépend d'une complexité de mise en œuvre et exige la spécification d'un constructeur et d'une méthode d'accès pour chaque dimension. Ainsi , La plupart des auteurs de bibliothèques d'algèbre linéaire fournissent des pseudo-arrays qui ciblent exclusivement des performances élevées avec une représentation de la dimensionnalité limitée. Pour une meilleure illustration de cette limitation, Blitz++ [4] limite la dimensionalité à onze dimensions, Armadillo [5] limite la dimisionnalité à trois dimensions, alors que Blaze [6], dlib [7], Eigen [8], cusp [9], et Elemental [10] limitent la dimensionnalité à deux dimensions.

2. Accès non-uniforme (No-Uniform Access Principal) :

   Un conteneur de données étant un objet qui enveloppe ses attributs et permet aux programmeurs d'y accader aux moyens de méthodes membres, en tant que tels, les conteneurs de données ne dérogent pas à la règle Uniform Access Principal (UAP)

[3] http://folk.uio.no/patricg/blitz/html/array-Impl_8h-source.html
[4] https://sourceforge.net/projects/blitz/
[5] http://arma.sourceforge.net/
[6] https://bitbucket.org/blaze-lib/blaze
[7] http://dlib.net/
[8] http://eigen.tuxfamily.org/index.php?title=Main_Page
[9] https://github.com/cusplibrary/cusplibrary
[10] http://libelemental.org/

introduite par Bertrand Meyer qui stipule que *Tous les services offerts par un module devraient être disponibles par une notation uniforme, indépendament de leurs mises en œuvre qu'elles soient par stockage ou par calcul*.

Toutefois, ce principe n'est pas respecté par les pseudo-arrays qui utilisent une technique d'accès par surcharge de l'opérateur () tandis que les tableaux POD utilisent la méthode d'accès de l'opérateur subscript[], introduisant ainsi une complexité supplémentaire et réduisant considérablement la cohabitation entre les tableaux POD et les conteneurs pseudo-arrays. À titre d'illustration, toutes les bibliothèques d'algèbre linéaire que nous avons étudiées comme Blitz ++ sont incompatibles avec les tableaux POD.

Pour une meilleure compréhension des implications directes du non respect UAP, nous considérerons le cas simple où nous effectuerons une opération de multiplication entre deux matrices A et B et stocker le résultat dans la matrice C (C = A $cdot$ B) que nous souhaitons généraliser aux tableaux de type (POD et Psuedo-Array). Nous serons alors amenés à écrire un programme pour PODs $[A : POD, B : POD, C : POD]$ } illustré dans le linsting 5.1 et un autre programme pour pseudo-arrays $[A : PArr, B : PArr, C : PArr]$ } illustré par le listing 5.2.

```
template <typename T, typename V, typename U> void MatMultPOD(T& C,V& A,U& B)
{
 for int i=0;i<..,i++
    for int j=0;j<..,j++
     c[i][j] += a[i][k] * b[k][j];
};
```

Listing 5.1 – Produit générique de deux matrices POD

.

```
7  template <typename T,typename V,typename U> void MatMultPArr(T& C,V& A,U& B)
8  {
9  for int i=0;i<..,i++
10     for int j=0;j<..,j++
11      c(i,j) += a(i,k) * b(k,j);
12 };
```

Listing 5.2 – Produit générique de deux matrices pseudo-arrays

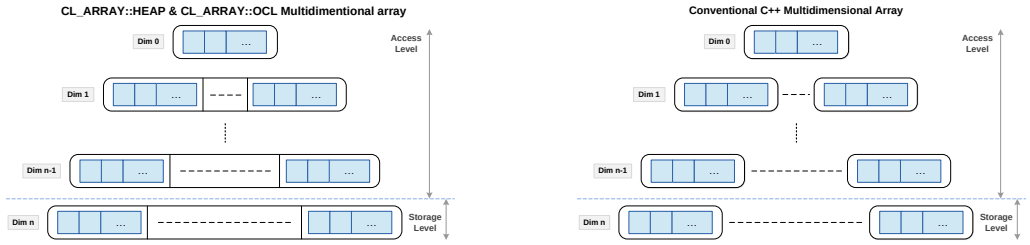
Dans le cas où nous effectuons une opération de multiplication entre deux matrices C = A · B de différents types (POD et PArray) dans le même programme, nous devrions écrire huit programmes différents, un pour chacune des combinaisons ci-dessous { $[A : POD, B : POD, C : POD]$ $[A : POD, B : POD, C : PArr]$ $[A : POD, B : PArr, C : POD]$ $[A : POD, B : PArr, C : PArr]$ $[A : PArr, B : POD, C : POD]$ $[A : PArr, B : POD, C : PArr]$ $[A : PArr, B : PArr, C : POD]$ $[A : PArr, B : PArr, C : PArr]$ }.

5.2.2.2 Notre proposition

Dans l'état actuel, les tableaux multidimensionnels dynamiques produits par la sémantique "vecteur à vecteur ... à vecteur", ou utilisant des bibliothèques alternatives telles que boost : : multiarray ou pseudo-arrays, ne conviennent pas pour fournir une sémantique multidimensionnelle unifiée de tableaux dynamiques tout en préservant un niveau de performance adapté aux applications parallèles.

Pour résoudre ce problème, nous proposons une alternative de tableaux multidimensionnels, que nous utilisons pour CL_ARRAY : :HEAP et CL_ARRAY : :OCL, comme indiqué dans le tableau 5.I et illustré par la figure 5.1. Ainsi , pour la représentation des tableaux multidimensionnels, nous proposons une forme de tableaux basée sur une structure arborescente, que nous appelons le *tableau multidimensionnel compact*, à Nd niveaux

dans l'optique de produire des structures avec une meilleure conscience du cache.



(a) Notre proposition de tableaux multidimentionnel compact

(b) Tableaux multidimentionnels C++ Conventionnels

Figure 5.1 – Les tableaux CL_ARRAY HEAP et OCL vs Tableaux C++ conventionnels

Plusieurs études [18, 31, 45, 46, 82, 95] confirment que les programmes basés sur des pointeurs souffrent souvent d'une mauvaise localité et sont difficiles à analyser et à interpréter par les compilateurs en raison de leur structures de données complexes. Pour réduire les défauts de cache des listes chaînées, trois méthodes sont souvent employées séparément ou d'une manière combinée : (i) Prélecture logicielle (Software prefetching), (ii) Prelecture matérielle (Hardware prefetching) et (iii) l'Amélioration de la localité spatiale des structures de données.

Dans ce contexte, Abdel Hameed Badawy et al. [9] ont utilisé CCMALLOC, un allocateur mémoire personnalisé, qui exploite une technique heuristique pour réserver des espaces mémoires adjacent pour les future allocations lors de l'attribution de nouveaux blocs de données [11].

En utilisant cet allocateur, les zones mémoires d'un même niveau sont plus susceptibles d'être regroupés dans des emplacements adjacents. Ainsi, CCMALLOC a permis aux auteurs d'améliorer à la fois les localités spatiales et temporelles des pointeurs par une forte réduction de la fragmentation des structures de données, réduisant la probabilité que des

[11] http://http://ccmalloc.sourcearchive.com

lignes de cache utiles soient expulsées du cache.

Pour le cas des tableaux multidimensionnels de style C ++ basé sur des structures chainées arborescentes, le nombre de dimensions correspondant à la profondeur de l'arbre et la relation entre les noeuds est toujours caractérisé par un chainage à pas constant. On pourrait donc s'attendre et dans le meilleur des cas à ce que les algorithmes heuristiques CCMALLOC appliqués à une matrice multidimensionnelle de N dimensions en C ++ conduisent à une structure arborescente de N dimensions compactes (une seule zone de mémoire par dimension).

La structure obtenue de cette façon est si simple qu'elle nécessite seulement un nombre d'allocations connues à la compilation égal au nombre de dimensions, évitant ainsi l'utilisation d'un allocateur mémoire personnalisé. A l'opposé des tableaux multidimensionnels dynamiques C++ A à N_d dimensions $[a_0]...[a_{N_d-1}]$, le nombre d'allocations mémoire requis est donné par la formule 5.1 :

$$Number_{OfAllocations} = Number_{OfNodes} = 1 + \sum_{i=0}^{N_d-2} (\prod_{j=0}^{i} (a_i)). \tag{5.1}$$

Alors que , la structure proposée ne nécessite que N_d allocations , comme exprimé par la formule 5.2 :

$$Number_{OfAllocations} = Number_{OfNodes} = N_d. \tag{5.2}$$

Tableau 5.I – Comparatif entre CL_ARRAY : :HEAP et CL_ARRAY : :OCL et les autres librairies

Conteneurs	**Methode d'accès**	**Stockage**	**Dimensions**	**Complexité**
boost : :multiarray	subscript[]	Non contigu	Pas de limite	$O(N^{Nd-1})$ (Formule 5.1)
vector of vector	subscript[]	Non contigu	Pas de limite	$O(N^{Nd-1})$ (Formule 5.1)
Pseudo-array	operator()	Contigu	limitée	O(1)
CL_ARRAY::HEAP	operator[]	Contigu	Pas de limite	$O(Nd)$
CL_ARRAY::OCL	operator[]	Contigu	Pas de limite	$O(Nd)$

Le passage d'une structure arborescente dispersée à une structure regroupée par dimension devrait en théorie réduire considérablement les défauts de caches. Pour mieux comprendre cette optimisation, nous allons examiner le comportement des processeurs modernes en présence des structures arborescentes, en effet les processeurs modernes sont tous équipés de Hardware Perfecther, qui a pour mission d'améliorer l'utilisation des caches internes des processeurs par l'application du processus suivant :

1. Analyse de la régularité des accès par des algorithmes heuristiques.
2. Prédit par des algorithmes heuristiques les segments de données en mémoire les plus susceptibles d'être utilisées dans le futur.
3. Transfert les données prédites de la mémoire centrale aux caches des processeurs.

Comme indiqué par l'étude proposée par Jaekyu Lee et al. [45], les analyses des accès des hardware perfectcher s'effectuent selon le type d'accès à la mémoire qui peut être soit directe (sans pointers et sans indirection pour le calcul des indexes) soit indirecte (indirection par des pointers ou par indirection de calcul des indexes) , pour la cas bien précis des accès directes aux tableaux réguliers le hardware perfectcher arrive le plus souvent a déduire simplement le modèle d'accès et améliore par conséquence convenablement l'utilisation des caches . À l'opposé, les accès indirectes par pointers le hardware perfectcher adopte une stratégie différente dite par stream , où pour chaque niveau d'indirection le hardware perfectcher applique une stratégie de pré-chargement [45].
Bien que les accès mémoires aux tableaux multidimensionnels sont souvent à pas constants, pour le cas bien précis des zones mémoires dispersées, le hardware perfectcher ne peut pas statuer sur le nombre d'indirections et aura du mal à prédire correctement les bloques mémoires occupés par les pointers les plus susceptible d'être utilisés dans le proche futur [9, 45, 71].
En outre, la stratégie proposée permet également d'aider le hardware perfectcher pour qu'il

puisse prendre les bonnes décisions et aboutir a des réductions significatives des défauts de caches , en effet , en présentant un nombre de stream égale au nombre d'indirection le hardware perfectcher peut statuer simplement sur le nombre d'indirections au bout de quelques défauts de caches et de prendre la décision adaptée pour chaque niveau de la structure de données multidimensionnels.

5.2.3 Nouveau formalisme pour la représentation de la dimensionnalité

5.2.3.1 Positionnement du problème

Les nouvelles versions d'OpenCL s'orientent de plus en plus vers un langage C ++ afin simplifier son utilisation et améliorer son interopérabilité avec d'autres plates-formes parallèles ou séquentielles Ainsi, il est important de veiller à ce que la nouvelle bibliothèque CL_ARRAY s'interface simplement avec les API d'OpenCL mais aussi avec d'autres bibliothèques C ++ standard. Dans ce contexte, malgré l'importance des tableaux multidimensionnels, les compilateurs C ++ souffrent d'un contexte formel pour la représentation des structures dimensionnelles [51, 59].

5.2.3.2 Notre proposition

Ce qui suit (dans le tableau 5.II) sont les descriptions des notations qui seront utilisées pour décrire le formalisme proposé.

Tableau 5.II – Notation

B_t : Type de base : $\triangleq$ Type natif $\triangleq Double|Float|..|Int$

Type complexe $\triangleq Struct|Union|Class$

N : Type naturel : $\triangleq int|size\_t$ $\{a_0, ..., a_{N_d-i}, ..., a_{N_d-1}\}$,$\{b_0, ..., b_{N_d-i}, ..., b_{N_d-1}\}$: are of type N

N_d est de type $N \triangleq$dimensionalité, i est de type $N < N_d$

T_{N_d} Type multidimentionnel POD $\triangleq \underbrace{B_t[a_0]\cdots[a_{N_d-1}]}_{N_d}$

T_{N_d-i} : Type multidimentionnel POD $\triangleq \underbrace{B_t[a_0]\cdots[a_{N_d-i}]}_{N_{d-i}}$

$NewT_{N_d}$: Nouveau type multidimentionnel$\triangleq \underbrace{B_t[b_0]\cdots[b_{N_{d-1}}]}_{N_d}$

PT_{N_d} : N_d Type pointer multidimentionnel$\triangleq \underbrace{B_t^{*\cdots*}}_{N_d}$

PT_{N_d-i} : $N_d - i$ Type pointer multidimentionnel$\triangleq \underbrace{B_t^{*\cdots*}}_{N_d-i}$

Ptr_{N_d}, Ptr_{N_d-i} : Pointer multidimentionnel (of type PT_{N_d}, PT_{N_d-i})

Pour surmonter le manque de représentation des conteneurs multidimensionnels des différentes normes C/C ++, nous proposons de représenter une dimensionnalité uniforme en définissant les règles de syntaxiques et sémantiques ci-dessous. La syntaxe et la sémantique adoptées exploitent les mêmes principes de typage statique proposés par Predrag et al. [76] :

1. $Container_{UnResizable} < T_{N_d} >$: Pour exprimer les conteneurs non redimensionnables.

2. $Container_{Resizable} < B_t, N_d >$: Pour exprimer les conteneurs redimensionnables.

La Table 5.III décrit la syntaxe abstraite et la sémantique des conteneurs CL_ARRAY conformément aux règles proposées.

Tableau 5.III – Types de conteneurs CL_ARRAY

CL_ARRAY	**Type de conteneur**	**Operations**
$Stack < T_{N_d} >$	$Container_{UnResizable}$	{Construction}
$Heap < T_{N_d}, Align = 0 >$	$Container_{UnResizable}$	{Construction,Liberation}
$OCL < T_{N_d} >$	$Container_{UnResizable}$	{Construction,Liberation}
$Heap < B_t, N_d >$	$Container_{Resizable}$	{Construction,Redimensionnement ,Insertion , Suppression,Liberation}

Align : Align est un paramètre template qui exprime l'alignement de pointeur lorsque l'alignement de la mémoire du pointeur est requis [55].

5.2.3.3 Structures unifiées et UAP

Les techniques de lecture et d'écriture non conventionnelles pour l'accès aléatoire aux éléments utilisés par un pseudo-array multidimensionnel influencent fortement sur la portabilité et la généricité du code utilisateur. Notre proposition, décrite dans le tableau 5.IV, conserve la sémantique multidimensionnelle POD du C/C ++.

Ainsi, tous les conteneurs proposés sont des tableaux flexibles accessibles par des appels récursifs d'opérateurs subscript [] et enroulant au moins deux attributs. Le premier attribut correspond aux données multidimensionnelles, tandis que le second attribut est un pointeur unidimensionnel qui pointe vers le premier élément du niveau de stockage.

Tableau 5.IV – Accès aléatoire aux éléments : CL_ARRAY

Conteneurs	**attributs**	**Methode d'accès**	**Retour**
$STACK$	$Data$ type of T_{N_d} Ptr_1 type of PT_1	$Operator()[i] \rightarrow \ \&(T_{N_d-1})$	Data[i]
$HEAP$ et OCL	$Data$ type of PT_{N_d} Ptr_1 type of PT_1	$Operator()[i] \rightarrow \&(PT_{N_d-1})$	Data[i]

5.2.3.4 Syntaxe et semantiques

Par l'application du formalisme de la dimensionnalité nouvellement proposé tel que décrit dans la section 5.2, nous obtenons la syntaxe et la sémantique présentée dans Table 5.V . Les conteneurs CL_ARRAY fournissent un moyen efficace et très intuitif d'exprimer les tableaux N-dimensionnels ainsi que les opérations appliquées à ces tableaux (redimensionnement, insertion, suppression ou libération) par rapport aux alternatives existantes. Par exemple, pour créer un tableau à trois dimensions utilisant boost : :multiarray, les programmeurs utilisent la syntaxe $boost :: multiarray < double, 3 >$ $(boost :: extents[2][3][4])$ ou bien $vector < vector < vector < double >>> (2, vector < vector < double >> (3, vector < double > (4)))$ ou $std :: array < std :: array < std :: array < double, 2 >, 3 >, 4 >$ pour générer le tableau alloué dans la mémoire heap.
Par contraste , en utilisant la bibliothèque CL_ARRAY, les programmeurs disposent d'une syntaxe très simple et intuitive, HEAP< $double[2][3][4] >$ et STACK < $double[2][3][4] >$, pour générer deux tableaux multidimensionnels alloués dans respectivement dans la mémoire tas et la mémoire de pile.

Tableau 5.V – Syntaxe et Sémantique

Syntaxe	**Sémantique**
STACK< T_{N_d} > *S0*	Création d'un tableau *S0* multidimensionnel alloué dans la mémoire pile
HEAP< $T_{N_d}, Al = 0$ > *S1*	Création d'un tableau *S1* multidimentionnel non redimensionnable alloué dans la mémoire du tas. Al est la valeur d'alignement , qui doit être une puissance de 2.
Heap< B_t, N_d > *S2*$(a_0, .., a_{N_d-1})$	Création d'un tableau *S2* multidimentionnel redimensionnable alloué dans le tas mémoire
OCL< T_{N_d} >*S3*(MapBuffer)	Création d'un tableau *S3* multidimentionnel non-redimensionnable alloué dans la mémoire du tas et la mémoire du dispositif OpenCL. MapBuffer est un paramèter template qui enveloppe les attributs CL_MapBuffer nécessaires pour exécuter les opérations clEnqueueMapBuffer.
S2=*CL_RESIZE*< $NewT_{N_d}$ > ()	Redimensionnement de *S2* avec la nouvelle dimensionalité < $NewT_{N_d}$ > (), si la dimensionalité est avant la compilation
S2=*CL_RESIZE*$(b_0, ..., b_{N_d-1})$	Redimensionnement de *S2* avec la nouvelle dimensionalité $(b_0, ..., b_{N_d-1})$ si la la dimensionalité est inconnue au compile time
S2=*CL_INSERT*(P, N, D)	Insertion danns S2 de M elements à la position P correspondante à la dimension D
S2=*CL_DELETE*(P, N, D)	Supression de S2 de M elements à la position P correspondante à la dimension D
S1,S1,S2=*CL_FREE*	Libération de la mémoire occupée par S1, S2 et S3

5.2.4 Stratégie d'allocation et utilisation OpenCL

Tous les conteneurs CL_ARRAY STACK, OCL et HEAP sont basés sur un stockage contigu, que nous appelons "niveau de stockage", mais avec différentes stratégies d'allocation. Le niveau de stockage se réfère à une zone de mémoire contiguë pour le stockage des données brutes selon les différentes politiques décrites dans le tableau 5.VII adapté à chaque catégorie de conteneur.
Les conteneurs CL_ARRAY HEAP et OCL utilisent les pointeurs multidimensionnels C. Nous appelons la mémoire occupée par les pointeurs multidimensionnels "niveau d'accès". Comme décrit dans le tableau 5.VI, le niveau d'accès est toujours alloué par les allocateurs système est structuré dans un ensemble des zones mémoire contiguë destinées au stockage des pointeurs multidimensionnels.

Tableau 5.VI – Hiérarchie mémoire des conteneurs CL_ARRAY

Conteneurs	**Niveau d'accès**	**Niveau de stockahe**	**Politique**	**Utilisation**
STACK	Non utilisé	Host au compile time	PL_{Stack}	Zero-Copy et Default Copy.
HEAP	Host (OS Allocator)	Host (OS Allocator)	PL_{Heap}	Zero-Copy et Default Copy.
OCL	Host (OS Allocator)	Host|Device (OpenCL Allocator)	PL_{OCL}	Host et device

Tableau 5.VII – Sémantique des politiques

Politique	**Signature**	**Sémantiques**
PL_{Stack}	(T_{N_d} &$Data$,PT_1 &Ptr_1)	Usage de la fonction Update1dPtr($Data$,PT_1) define dans la table 5.IX
PL_{Heap} et PL_{OCL}	(PT_{N_d} &$Data$,PT_1 &Ptr_1)	(.) Alloue le niveau accès pointé par Data (.) Alloue le niveau de stockage pointé par Ptr_1

5.3 Algorithmes et complexité

La section suivante décrit les fonctions metaprogrammées statiques et dynamiques (la Table 5.VIII présente les fonctions statiques alors que la Table 5.IX présente les fonctions dynamiques) que nous avons définies et implémentées pour la conception et la mise en oeuvre des algorithmes originaux utilisés par la librarie CL_ARRAY.

Tableau 5.VIII – Compile-time métafonction

Metafonction	**Sémantique**
sizeof(T_{N_d})↦ N_{Byte} sizeof(B_t)↦ N_{Byte}	Retourne en byte , la taille mémoire resquise par T_{N_d}
Get$_{POD}$(T_{N_d},$i \in [0, N_d - 1]$)↦ T_{N_d-i}	Renvoie un type POD multidimensionnel qui correspond à la dimension i du type multidimensionnel T_{N_d} ; EX :, Get$_{POD}$ < $double[10][20][30], 1$ > Renvoie $double[10][20]$
Get$_{Pointer}$(T_{N_d},$i \in [0, N_d - 1]$)↦ PT_{N_d-i}	Renvoie un type de pointeur multidimensionnel qui correspond à la dimension i du type multidimensionnel T_{N_d}
Get$_{Rank}$(T_{N_d})↦ N,Get$_{Rank}$(PT_{N_d})↦ N	Récupère le nombre de dimensions du type multidimensionnel TN_d corrsespondant à T_{N_d} ou le pointeur multidimentionnel de type PT_{N_d}
Get$_{Extent}$(T_{N_d}, $i \in [0, N_d - 1]$)↦ N	Récupère, la valeur a_i du type multidimentionnel T_{N_d} correspondant a la dimension i. { $(i = 0 \Rightarrow Return\ a_0), ..., (i = N_d - 1 \Rightarrow Return\ a_{N_d-1})$}
Get$_{NbrOfElems}$(T_{N_d}, $i \in [0, N_d - 1]$)↦ N	Récupère le nombre d'éléments du type multidimensionnel T_{N_d} correspondant à la dimension i. Renvoie {a_0 if $(i = 0)$, $otherwise \prod_{k=0}^{i} Get_{Extent}(T_{N_d}, k)$}
Get$_{Size}$(T_{N_d}, $i \in [0, N_d - 1]$)↦ N	Size=Get$_{NbrOfElems}$(T_{N_d}, i)*sizeof(Get$_{Pointer}$,(T_{N_d},i))
Remove$_{Pointer}$(PT_{N_d}, $i \in [0, N_d - 1]$) ↦ PT_{N_d-i}	Renvoie un pointeur de type PT_{N_d-i}
Built$_{Pointer}$(B_t, N_d) ↦ PT_{N_d}	Renvoie un pointeur de type PT_{N_d}

Tableau 5.IX – Runtime métafonction

Fonction générique	**Sémantique**
Update1dptr ($\underbrace{Data}_{\text{is type of } T_{N_d}}$, Ptr_1)	Met à jour Ptr_1 par la reference du premier element du niveau de stockage $Ptr_1 \leftarrow\!\shortmid\ \& \underbrace{Data[0]...[0]}_{N_d}$
Allocate(T_{N_d},$i \in [0, N_d - 1]$) $\mapsto PT_{N_d-i}$	Alloue un bloc mémoire d'une taille en byte =Get$_{Size}$(T_{N_d},i) et renvoie un de type Get$_{Pointer}$(T_{N_d},i) qui pointe sur le début du bloc.
ReAllocate(Ptr_{N_d-i},T'_{N_d},$i \in [0, N_d - 1]$)	Realloue un bloc mémoire d'une taille en byte =Get$_{Size}$(T_{N_d},i) pointé par Ptr_{N_d-i}.
Release(Ptr_{N_d-i},$i \in [0, N_d - 1]$)	Libère le bloc mémoire pointé par Ptr_{N_d-i}.
MemMove($\underbrace{Ptr_{1^S}}_{\text{is type of } B_t^*}$, $\underbrace{Ptr_{1^D}}_{\text{is type of } B_t^*}$, $Size_{Byte}$)	Copie les valeurs de $Size_{Byte}$ bytes de l'emplacement pointé par Ptr_{1^s} vers le bloc mémoire pointé par Ptr_{1^d}.
$Create_{Tag}$(T_{N_d},tag)	Alloue Tag et le met à jour par $Update_{Tag}$(T_{N_d},tag),
$Update_{Tag}$(T_{N_d},tag)	Met à jour Tag par T_{N_d}, $\{Tag[0] = Get_{Extent}(T_{N_d}, 0), ..., Tag[N_d - 1] = Get_{Extent}(T_{N_d}, N_d - 1)\}$
$Ptr_{N_d}[a_i] \mapsto PT_{N_d-1}$	Renvoie la référence de l'élément de type PT_{n_d-1} à l'emplacement spécifié par a_i.

5.3.1 Construction de HEAP et OCL

5.3.1.1 Algorithme pour des matrices à deux dimensions

Avant de présenter l'algorithme générique, adopté pour la construction des conteneurs multidimensionnels du CL_ARRAY, nous allons examiner et comparer les deux programmes C suivants, pour la construction des conteneurs à deux dimensions [X][Y] de type double.

```
1 double** Array0;
2 Array0=(double**) malloc(X*sizeof(double*));
3 for(int i = 0;i< X;i++)
4    Array0[i]=(double*) malloc(Y*sizeof(double));
```

Listing 5.3 – Programme pour la construction d'une matrice à deux dimensions conventionnelle

```
1 double** Array1;
2 double r, *rp;
3 Array1 =(double**) malloc( X * sizeof (*Array1) );
4  for (  r = 0, rp = Array1[0] ; r < X ; r++, rp += Y )
5     Array1[r]=rp;
```

Listing 5.4 – Programme pour la construction d'une matrice à deux dimensions compacte

En dépit de la représentation spatiale qui différencie les deux structures Array0 et Array1 , produites respectivement par le linsting 5.3 et le linsting 5.3.
Array0, a nécessité X+1 allocations mémoire et autant d'opérations de mise à jour, tandis Array1 n'a nécessité que deux allocations mémoire et X opérations de mise à jours.
La réduction du nombre d'allocations mémoire, permet en théorie un gain en temps et en mémoire. En effet, chaque invocation pour une allocation mémoire est couteuse en temps et alloue le plus souvent, plus de mémoire que nécessaire [31].

5.3.1.2 Généralisation

Une généralisation du concept à deux dimensions et comme indiqué dans Section 5.2, CL_ARRAY :HEAP et CL_ARRAY :OCL sont des structures similaires des arborescences avec N_d niveaux hiérarchiques. La profondeur de cette hiérarchie est égale au nombre de dimensions. À la base de cette hiérarchie se trouve un pseudo-array contigu unidimensionnel. La figure 5.2 compare la structure d'un tableau tridimensionnel construit par Algorithm 1 à la structure de tableau C ++ classique construite par boost : :multiarray ou $std :: vector << std :: Vecteur < ... >>$.

5.3.1.3 Complexité

D'une manière générale, un conteneur mémoire A clairsemée de N dimension [a0,....an-1], nécessite $1 + a0 + (a0 * a1) + (a0 * a1 * a2) + ... + (a0 * a1 * .. * an - 2)$ (formule 5.1) , tandis qu'un conteneur CL_ARRAY HEAP ou OCL équivalent, ne nécessite que N opérations d'allocations Nd allocations (formule 5.2)

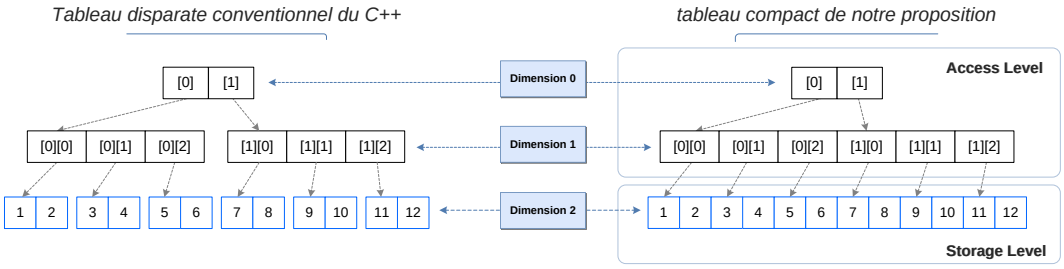


Figure 5.2 – Comparaison de la représentation spatiale des conteneurs tridimensionnels $[2][3][2]$. L'allocation de politique utilisée par les tableaux conventionnels C ++ et la politique d'allocation compacte utilisée pour CL_ARRAY : :OCL et CL_ARRAY : :HEAP.

5.3.1.4 Pseudo-code

Dans cette section, nous proposons un nouvel algorithme générique méta-programmé 1 pour produire les conteneurs de données CL_ARRAY : :HEAP et CL_ARRAY : :OCL à N_d dimensions.

```
1  Function AllocateContainer (T_{N_d}, Ptr_{N_d-step}, step, N_d)
2      Ptr_{N_d-step} ← Allocate(T_{N_d}, step);
3      step ← step + 1;
4      if step <= Nd − 1 then
5          AllocateContainer(T_{N_d}, Ptr_{N_d}[0], step, N_d);
6  Function UpdateContainer (T_{N_d}, Ptr_{N_d-step}, step, N_d)
7      loop_{interval} ← Get_{ext}(T_{N_d}, step);
8      loop_{condition} ← Get_{ext}(T_{N_d}, step + 1);
9      for (i ← loop_{interval}, j ← 1; i <= loop_{condition}; i ← i + loop_{interval}, j ← j + 1) do
10         Ptr_{N_d-step}[j] ← Ptr_{N_d-step}[0] + i;
11     step ← step + 1;
12     if step <= N_d − 2 then
13         UpdateContainer(T_{N_d}, Ptr_{N_d-step}[0], step, N_d);
14 Function BuildMultidimensionalContainer (T_{N_d})
15     Get_{Pointer} (T_{N_d},0) Ptr_{N_d} ;
16     int* tag ;
17     N_d ← Get_{Rank}(T_{N_d});
18     AllocateContainer(T_{N_d}, Ptr_{N_d}, 0, N_d);
19     if N_d > 1 then
20         UpdateContainer(T_{N_d}, Ptr_{N_d}, 0, N_d);
21     Create_{Tag}(T_{N_d},tag);
```

Algorithm 1: Construction des conteneurs de données multidimentionnels HEAP et OCL

5.3.2 Redimentionnement des conteneurs CL_ARRAY : :HEAP

5.3.2.1 Sémantique

La sémantique du redimensionnement équivaut à celle proposée par les bibliothèques multiarray et Blitz ++. la figure 5.3 illustre un exemple de redimensionnement d'un conteneur à deux dimensions.

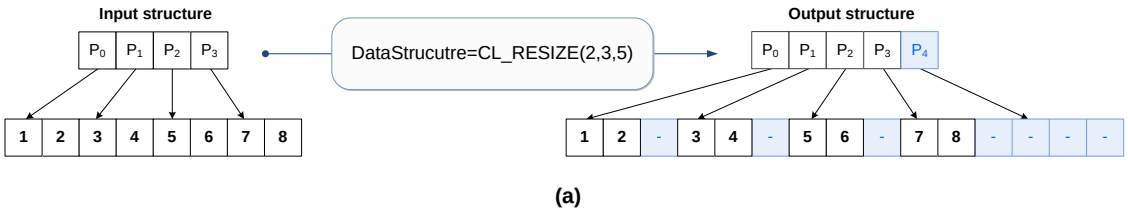


Figure 5.3 – Exemple de redimentionnement

5.3.2.2 Pseudo-code

Les conteneurs CL_ARRAY se caractérisent par un niveau d'accès contigu. Les opérations de redimensionnement aléatoire impliquant des réductions ou des augmentations du nombre d'éléments dans une ou plusieurs dimensions exigent, en plus des opérations de réalloctations mémoire, d'autres opérations de déplacement des données. Toute opération impliquant un mouvement de données nécessite une série de copies de mémoire, qui sont des opérations non seulement chronophage, mais nécessitent également l'usage d'une mémoire supplémentaire.
Pour réduire les risques de saturation mémoire, les nouveaux algorithmes génériques métaprogrammés 2,3,4 et 5,6 sont proposés pour le redimensionnement. Contrairement aux approches classiques, ils exploitent le principe de copies sur place (sans mémoire intermédiaire) :

- Si les nouvelles dimensions impliquent pour une ou plisieurs dimensions l'augmentation du nombre d'éléments (montré dans Algorithm 2), de n'importe quelle dimension, le rang de la dimension étant supérieur à l'ancien nombre d'éléments de cette même dimension, nous procédons a des déplacements par ordre décroissant (de bas en haut).

```
1 Function Copy_Decreasing(Ptr_N1, Nbr_OfElems, Old_Extent1, New_Extent1)
2     Nbr_OfCopy ← Nbr_OfElems/New_Extent1;
3     for (i ∈ [0, Nbr_OfCopy − 2]) do
4         loop_Dest ← Nbr_OfElems − (New_Extent1 ∗ (i + 1));
5         loop_Src ← (Nbr_OfCopy ∗ Old_Extent1) − (New_Extent1 ∗ (i + 1));
6         MemMove(Ptr_N1 + loop_Dest, Ptr_N1 + loop_Src, New_Extent1 ∗ sizeof(B_t));
```

Algorithm 2: Déplacement des données par ordre décroissant pour augmenter le nombre d'éléments

- Si les nouvelles dimensions impliquent pour une ou plusieurs dimensions la réduction du nombre d'éléments (montré dans Algorithm 3), de n'importe quelle dimension, le rang de la dimension étant inférieur à l'ancien nombre d'éléments de cette même dimension, nous procédons à des déplacements par ordre croissant (de haut vers le bas).

```
1 Function Copy_Increasing(Ptr_N1, Nbr_OfElems, Old_Extent1, New_Extent1)
2     Nbr_OfCopy ← Nbr_OfElems/New_Extent1;
3     for (i ∈ [0, Nbr_OfCopy − 2]) do
4         loop_Dest ← New_Extent1 ∗ (i + 1);
5         loop_Src ← Old_Extent1 ∗ (i + 1);
6         MemMove(Ptr_N1 + loop_Dest, Ptr_N1 + loop_Src, Old_Extent1 ∗ sizeof(B_t));
```

Algorithm 3: Déplacement des données par ordre croissant pour réduire le nombre d'éléments

Théoriquement, une opération de redimensionnement aléatoire pour plusieurs dimensions se décompose naturellement en N_d opérations de redimensionnement (dimension par dimension) ; Ces opérations peuvent être exécutées dans n'importe quel ordre. En outre, les opérations de redimensionnement nécessitant un nombre accru d'éléments nécessitent des opérations de réallocation mémoire antérieures au même niveau d'accès.

Pour réduire le nombre d'opérations de réallocation mémoire, nous organisons les opérations de déplacement de données en deux phases : la première phase vise à réduire le nombre d'opérations de réallocations, suivi d'une phase destinée à réduire le nombre d'opérations de copies.

- Algorithme 4 réductions d'opérations de copies par ordre croissant(de la dimension N à la dimension ($Rank$ = N_d-1).

Function $DecreasingElements(Ptr_{N_1}, Rank, Old_{tag}, New_{tag})$
  $OldNbr_{OfElems} \leftarrow \prod_{j=0}^{Rank-1}(Old_{tag}[j])$, $step \leftarrow Old_{tag}[0]$;
  for $(i \in [1, Rank-1])$ **do**
    if $Old_{tag}[i] > New_{tag}[i]$ **then**
      $New_{Extent} \leftarrow OldNbr_{OfElems}/step$; $Old_{Extent} \leftarrow (New_{Extent}/Old_{tag}[i]) * New_{tag}[i]$;
      $Copy_{Increasing}(Ptr_{N_1}, OldNbr_{OfElems}, New_{Extent}, Old_{Extent})$;
      $OldNbr_{OfElems} \leftarrow OldNbr_{OfElems} * New_{tag}[i]$;
    $step \leftarrow step * Old_{tag}[i]$;

Algorithm 4: Réduction du nombre de copies

- Algorithm 5 Effectue les opérations d'augmentation de données par ordre décroissant.

Function $IncreasingElements(Ptr_{N_1}, Rank, Old_{tag}, New_{tag})$
  $OldNbr_{OfElems} \leftarrow \prod_{j=0}^{Rank-1}(Old_{tag}[j])$, $step \leftarrow New_{tag}[Rank-1]$;
  for $(i \in [1, Rank-1])$ **do**
    if $Old_{tag}[i] < New_{tag}[i]$ **then**
      $OldNbr_{OfElems} \leftarrow (OldNbr_{OfElems}/Old_{tag}[i]) * New_{tag}[i]$;
      $New_{Extent} \leftarrow step$; $Old_{Extent} \leftarrow (New_{Extent}/New_{tag}[i]) * Old_{tag}[i]$;
      $Copy_{Decreasing}(Ptr_{N_1}, OldNbr_{OfElems}, New_{Extent}, Old_{Extent})$;
      $Old_{Tag}[Rank-i] \leftarrow New_{Tag}[Rank-i]$;
    $step \leftarrow step * New_{tag}[Rank-i-1]$;

Algorithm 5: Redimensionnement pour l'augmentation du nombre des éléments

Ainsi (Algorithme 6),et à l'appui de l'organisation proposée en deux phases , on optimise le nombre requis de réallocations mémoires (Number reallocation = N_d) indépendamment de la nature du redimensionnement requis.

Function ReAllocateDataContainer ($T_{N_d}, Ptr_{N_d-step}, step, N_d$)
    $Ptr_{N_d-step} \leftarrow ReaAllocate(Ptr_{N_d}, T_{N_d}, step)$;
    $step \leftarrow step + 1$;
    if $step <= Nd - 1$ **then**
        $ReAllocateDataContainer(T_{N_d}, Ptr_{N_d-step}[0], step, N_d)$;

Function ReallocateMultidimensionalContainer ($NewT_{N_d}, Ptr_{N_d}$)
    $int^* \ New_{tag}$; $int^* \ Old_{tag} \leftarrow tag$;
    $Create_{Tag}(NewT_{N_d}, New_{tag})$;
    $N_d \leftarrow Get_{Rank}(NewT_{N_d})$;
    $OldNbr_{OfElems} \leftarrow \prod_{j=0}^{Rank-1}(Old_{tag}[j])$;
    $NewNbr_{OfElems} \leftarrow \prod_{j=0}^{Rank-1}(New_{tag}[j])$;
    $B_t^* \ Ptr_{N_1} \leftarrow \underbrace{Ptr_{N_d}[0]..[0]}_{N_d-1}$;
    `// any change in dimension 0, does not require data movement`
    if $New_{tag}[0] > Old_{tag}[0]$ **then**
        $New_{tag}[0] \leftarrow Old_{tag}[0]$;
    else
        $Old_{tag}[0] \leftarrow New_{tag}[0]$;
    if $NewNbr_{OfElems} < OldNbr_{OfElems}$ **then**
        $DecreasingElements(Ptr_{N_1}, Rank, Old_{tag}, New_{tag})$;
        $IncreasingElements(Ptr_{N_1}, Rank, Old_{tag}, New_{tag})$;
        $ReAllocateDataContainer(NewT_{N_d}, Ptr_{N_d}, 0, N_d)$;
    else
        $ReAllocateDataContainer(NewT_{N_d}, Ptr_{N_d}, 0, N_d)$;
        $DecreasingElements(Ptr_{N_1}, Rank, Old_{tag}, New_{tag})$;
        $IncreasingElements(Ptr_{N_1}, Rank, Old_{tag}, New_{tag})$;
    $Create_{Tag}(NewT_{N_d}$,tag);

Algorithm 6: Redimensionnement d'un conteneur multidimensionnel CL_ARRAY

.

5.3.2.3 Complexité

Pour redimensionner un conteneur C++ conventionnel produit par la sémantique vecteur de vecteur ...de vector, aux dimensions $[a_0, ...a_{N_d-1}]$ avec de nouvelles dimensions $[b_0]...[b_{N_d-1}]$ le pire cas s'obtient pour $(b_0 > a0, .., b_{N_d} > a_{N_d})$. Au plus cette opération de

redimensionnement nécessite :

$$\begin{cases} Number_{OfReAllocations} = 1 + \sum_{i=0}^{N_d-2}(\prod_{j=0}^{i}(a_i)) \\ \qquad\qquad ET \\ Number_{OfAllocations} = \sum_{i=0}^{N_d-2}(\prod_{j=0}^{i}(b_i) - \prod_{j=0}^{i}(a_i)) \end{cases} \tag{5.3}$$

En revanche, l'utilisation de l'algorithme proposé pour redimensionner les conteneurs contigus nécessite uniquement :

$$\begin{cases} Number_{OfReAllocations} = N_d \\ \qquad\qquad ET \\ Number_{OfCopy} = (\sum_{i=1}^{N_d-2}(\prod_{j=0}^{i}(a_j))) - N_d \end{cases} \tag{5.4}$$

5.3.3 Insertion et suppression des éléments

Comme illustré par les figures 5.4 et 5.5 ,les opérations suivantes sont prévues pour insérer ou supprimer des éléments M à la position $P \in [0, Extent(T_{N_d}, D)]$ dans la dimension $D \in [0, N_d - 1]]$ [12].

Les algorithmes proposés pour l'insertion et la suppression d'éléments sont semblables à ceux proposés pour augmenter ou réduire le nombre d'éléments ($Copy_{Decreasing}$) et ($Copy_{Increaseasing}$). Les différences se situent dans les calculs de la source et de la destination pour chaque opération de copie.

Pour maintenir la cohérence des données, pour le cas de l'insertion, les opérations de réallocation précédent les opérations de copie ; En revanche, pour les opérations de suppression de données, les opérations de copie sont excutées avant les opérations de réaffectation.

[12] pour une opération de suppression $M \in [0, Extent(T_{N_d}), D]$

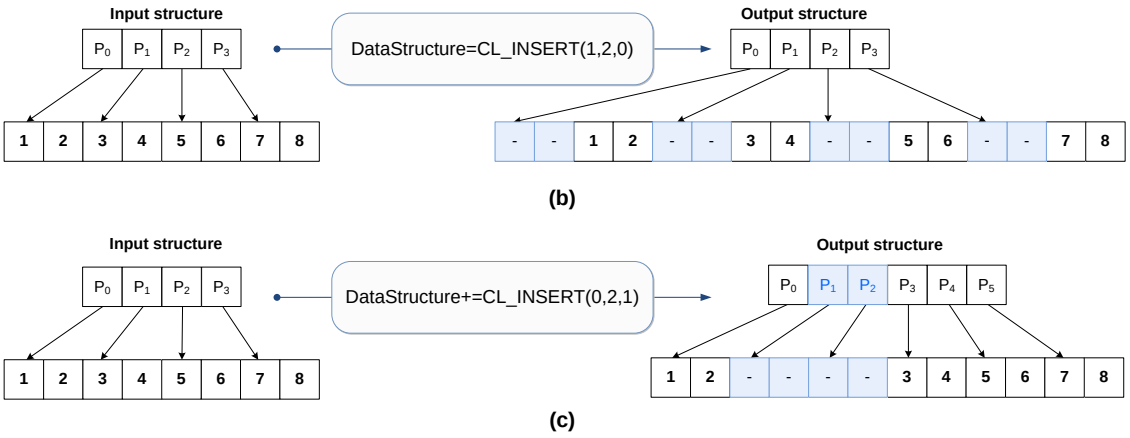


Figure 5.4 – Examples : sémantique d'insertion des éléments

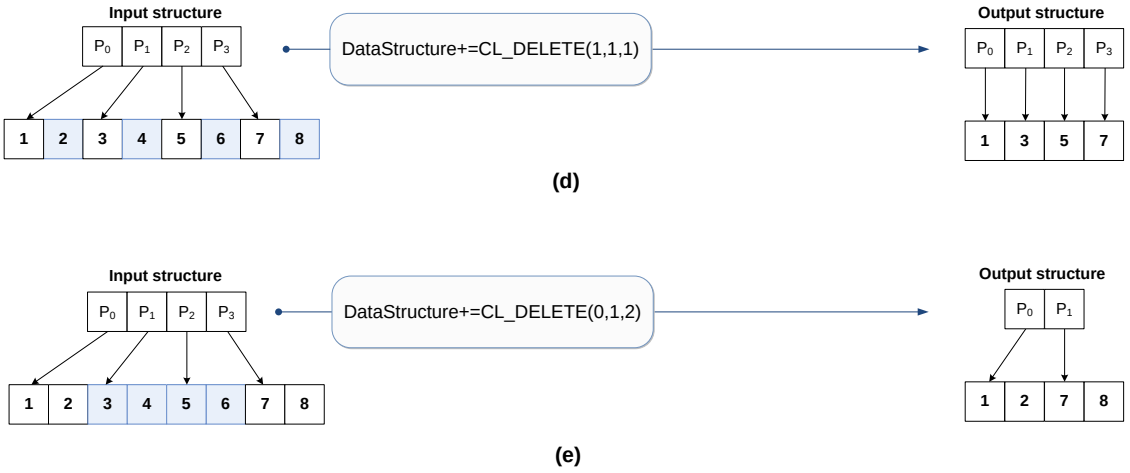


Figure 5.5 – Examples : sémantique de suppresion des éléments

5.3.4 Libération mémoire

Chaque bloc mémoire alloué doit être libéré. Par conséquent, le nombre d'opérations requis pour la libération de la mémoire occupée par un conteneur de données est le même requis pour les opérations de construction du même conteneur de données. Ainsi, on obtient N_d opération de désallocation mémoire pour les conteneurs CL_ARRAY et le nombre

donné par la formule 5.1 pour les conteneurs C++ conventionnels.

5.4 OCL_ARRAY_VIEW

Les structures de données multidimensionnelles sont fondamentalement d'une dimentionalité quelconque, alors que le modèle d'exécution OpenCL est soit à une, à deux ou à trois dimensions (selon la configuration du NDrange). Ainsi, il est nécessaire de fournir des transformations multidimensionnelles représentatives du modèle d'exécution et adaptée au techniques d'équilibrage de charge et / ou de dépendances.
Le choix d'un niveau d'accès contigu nous permet d'offrir une solution pour exprimer ces transformations multidimensionnelles. Notre approche repose donc sur des structures objets enveloppés au niveau de stockage en poupées russes, ce qui permet des vues multidimensionnelles partielles ou totales qui représentent le modèle d'exécution, de partage de charge et / ou des dépendances.

Les vues multidimensionnelles, sont des classes modèles qui représentent les données multidimensionnelles d'un type B_t, de M dimensions spécifiées. Au moment de la compilation, si M n'est pas spécifiée, il est supposé qu'il s'agit d'une vue à une seule dimension.
Les vues multidimensionnelles permettent de manipuler l'un des trois conteneurs CL_ARRAY {STACK,HEAP ou OCL} à N dimensions autant que structure à M dimensions, si la condition suivante est satisfaite : Nombre total d'éléments indiqué par la structure enveloppante (abstraite) doit être inférieur ou égal au nombre d'éléments de la structure enveloppée ((STACK,HEAP, ou OCL)).

La figure 5.6 illustre le principe de fonctionnement d'OCL_ARRAY_VIEW à l'appui d'un exemple de structure à deux dimensions $[a_0 = 4][a_1 = 6]$, a laquelle nous appliquons une vue tridimensionnelle (à trois dimensions) $[v_0 = 2][v_1 = 3][v_2 = 4]$. Les itérateurs *begin()* et *end()* nous permettent de définir les limites pour chaque appel de l'opérateur subscript

[], tandis que la méthode *GetNbrElm()* récupère le nombre d'éléments.

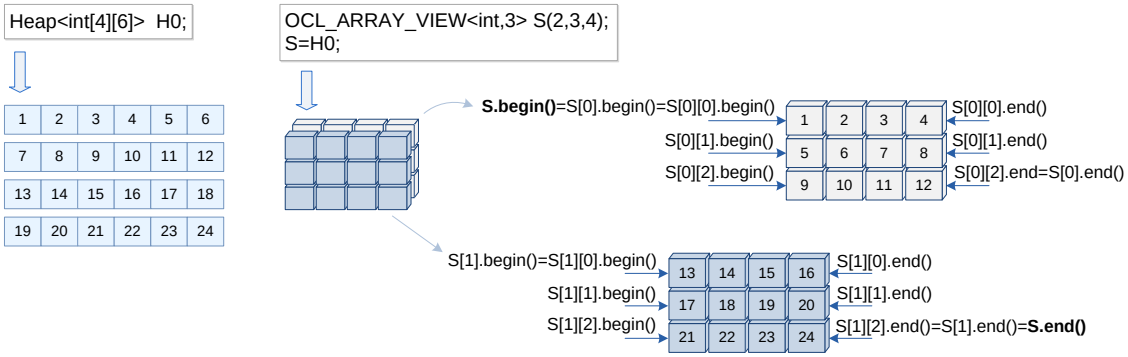


Figure 5.6 – Exemple : OCL_ARRAY_VIEW

5.4.1 Fonctionalité d'OCL_ARRAY_VIEW

Les conteneurs CL_ARRAY associent à une sémantique strictement régulière une mémoire d'accès contigüe, ils permettent ainsi au moyen d'un formalisme, la mise en correspondance entre la structure multidimensionnelle des conteneurs et le modèle de dépendance. Il advient, qu'un utilisateur peut expérimenter différentes sémantiques de dépendances sans apporter de modifications structurelles aux conteneurs ou de prévoir des conteneurs intermédiaires ; cette association permet également, aux concepteurs de DESL, de formaliser des combinaisons sémantique des modèles de dépendances pour concevoir des modèles de programmations évolués : Squelettes algorithmiques , programmation parallèle structurée , Array Based Programming ..ETC.).

En effet, un utilisateur OpenCL a besoin d'au moins deux fonctionnalités pour appeler l'API OpenCL clCreateBuffer :

1. Un pointeur unidimentionnel sur le premier élément du niveau d'accès des conteneurs CL_ARRAY

2. Un scalaire indiquant la taille du bloc mémoire (en octets) qui represente la quantité de données qui sera copiée ou partagée avec le périphérique OpenCL.

Ces deux fonctionnalités sont fournies respectiviment par les méthodes GetPointer() et GetSizeOfElems(). La structure OCL_ARRAY _VIEW fournit, en plus de ces deux fonctionalités de base

1. une sémantique complète des transformations multidimensionnelles ;
2. une nouvelle sémantique d'itérateurs multidimensionnels permettant l'utilisation de conteneurs CL _ARRAY dans des environnements parallèles ou séquentiels nécessitant des iterateurs begin(), end (), rebegin () ou rend (), tels que les algorithmes de la STL , les APIs SYCL (SYCL : :buffer), **parallel For** d'OpenMP 3.0 ou **parallel_for** d'Intel TBB..

5.4.2 Sémantique des itérateurs multidimentionnels

Pour les itérateurs d'OCL_ARRAY_VIEW, nous proposons la sémantique suivante :
Soit S vue abstraite multidimensionnelle à la dimensionnalité M_d et aux dimensions abstraites $[V_0]..[V_{Md}-1]$ appliquée un conteneur CL_ARRAY à la dimensionalité N_d et aux dimensions $[a_0]..[a_{Nd}-1]$
Sémantique de l'itérateur begin() et GetPtr() :

$$S.GetPtr() \triangleq S.Begin() \triangleq Ptr \leftarrow \&S\underbrace{\overbrace{[0]...[0]}^{M_d}}_{LSD}$$

$$S[ind_0].GetPtr() \triangleq S[ind_0].Begin() \triangleq Ptr \leftarrow \&S\overbrace{\underbrace{[ind_0]}_{\text{MSD}}\underbrace{[0]...[0]}_{LSD}}^{M_d}$$

$$S\overbrace{[ind_0]..[ind_{\eta-1}]}^{\eta<M_d}.GetPtr() \triangleq S\overbrace{[ind_0]..[ind_{\eta-1}]}^{\eta}.Begin() \triangleq Ptr \leftarrow \&S\overbrace{\underbrace{[ind_0]..[ind_{\eta-1}]}_{\text{MSD}}\underbrace{[0]...[0]}_{LSD}}^{M_d}$$

$$S[ind_0]..[ind_{M_d-1}].GetPtr() \triangleq S[ind_0]..[ind_{M_d-1}].Begin() \triangleq Ptr \leftarrow \&S\underbrace{\overbrace{[ind_0]..[ind_{M_d-1}]}^{M_d}}_{\text{MSD}}$$

Semantic de la méthode GetNbrOfElems() :

$$S.GetNbrOfElems() \leftarrow \prod_{j=0}^{M_d-1}(V_j)$$

$$S[ind_0].GetNbrOfElems() \leftarrow \prod_{j=1}^{M_d-1}(V_j)$$

$$S[ind_0]..[ind_{\eta-1}].GetNbrOfElems() \leftarrow \prod_{j=\eta}^{M_d-1}(V_j)$$

$$S[ind_0]..[ind_{M_d}-1].GetNbrOfElems() \leftarrow 1$$

Sémantique de l'itérateur End()

$$S.End()) \triangleq S.Begin() + S.GetNbrOfElems() - 1$$

$$S[ind_0]..[ind_{\eta-1}].End() \triangleq S[ind_0]..[ind_{\eta-1}].Begin() + S[ind_0]..[ind_{\eta-1}].GetNbrOfElems() - 1$$

$$S[ind_0]..[ind_{M_d-1}].End() \triangleq S[ind_0]..[ind_{M_d-1}].Begin()$$

5.4.3 Structure objet d'OCL_ARRAY_VIEW

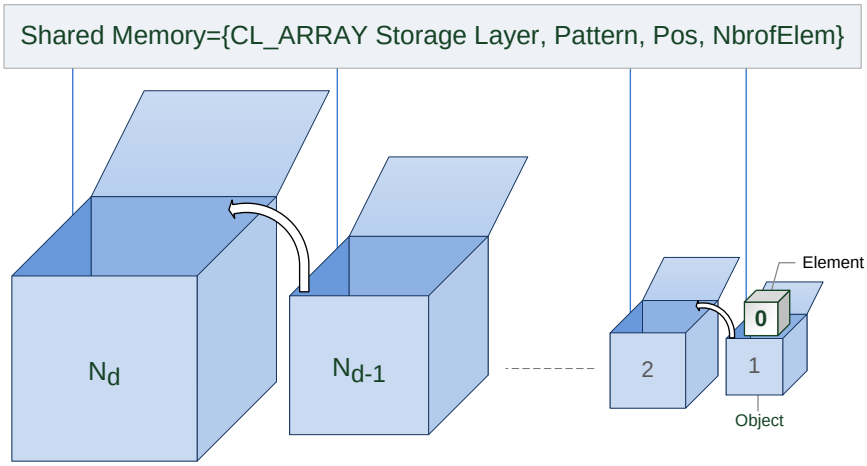


Figure 5.7 – Structure objet d'OCL_ARRAY_VIEW

Pour obtenir cette sémantique, comme illustrée dans la figure 5.7, nous proposons une structure OCL_ARRAY_VIEW en poupées russes "Russian Doll". Ainsi, l'objet OCL_ARRAY_VIEW M-dimensionnel contient un membre objet OCL_ARRAY_VIEW M-1 multidimentionnel. De même, l'Objet membre contient un autre objet OCL_ARRAY_VIEW N-2-dimensionnel, et ainsi de suite , jusqu'à ce que nous atteignions le dernier objet à une seule dimension, qui ne contient aucun autre objet.

Les ressources mémoire sont partagées entre tous les objets et peuvent être considérées comme N_uplet :La couche de stockage de la structure CL_ARRAY (HEAP, STACK et OCL) ; Le modèle qui stocke la dimensionnalité ; La position de l'index ; Le compteur NbrOfElements ; Et un pointeur unidimensionnel.

À chaque appel d'opérateur subscript [], la position de l'index et le nombre d'éléments sont mis à jour. Chaque objet membre est doté d'une méthode d'accès par la surcharge de l'opérateur subscript [], qui renvoie une référence à son objet membre de la dimension inférieure - sauf pour le dernier objet, qui renvoie une référence sur l'élément.

5.5 Conclusion

Le manque constant d'une sémantique concrète pour la réprésentation des systèmes multidimentionnels était au coeur de nos motivations , et nous avons jugé utile de reconsidérer tous les aspects conceptuels liés aux ingénieries des conteneurs (permettant la génération, l'exploitation et la gestion des structures de données multidimensionnelles), et proposer une approche conceptualisée spécifique aux environnements OpenCL.

Dans ce chapitre , nous avons présenté les opportunités majeures offertes par CL_ARRAY et OCL_ARRAY_VIEW pour l'écosystème OpenCL en termes de simplification, de sécurisation et d'optimisation.

Le chapitre 6 , est prévu dans l'optique d'apporter des précisions relatives à l'implémentation de la librairie CL_ARRAY , quelques exemples triviaux qui illustrent son utilisation ainsi que son évaluation dans le contexte de trois applications d'algèbre multilinéaire et de traitement des images à deux dimensions.

CHAPITRE 6

IMPLÉMENTATION , APPLICATIONS ET ÉVALUATIONS

6.1 Introduction

L'avantage majeur des tableaux multidimensionnels par rapport aux autres structures de données est qu'un programmeur peut les utiliser avec des accès aléatoires ; cependant l'implémentation naïve des tableaux se réduit à un pointeur unidimensionnel vers une zone mémoire contigüe allouée dynamiquement, et une méthode d'accès qui émule par calcul d'indices le comportement de structures de données multidimensionnelle. Comme précisé dans le chapitre 5 , ce type de pratique limite considérablement l'évolutivité et la généricité des programmes.
La conception et la mise en œuvre de conteneur de données est à ce jour un problème ouvert , A cet effet, nous citons la réflexion de l'inventeur du C++ Bjarne Stroustrup dans le chapitre 29 de son ouvrage culte [85] , ***I hav e never seen a perfect matrix class. In fact, given the wide variety of uses of matrices, it is doubtful whether one could exist***. Ainsi , l'expérience CL_ARRAY à pour ambition de proposer de nouvelles conceptions et implémentations plus adaptées pour les applications scientifiques par de nouveaux apports sémantique , syntaxique et algorithmiques.
Dans ce chapitre, nous présentons en premier lieu les techniques majeures qui ont mené à l'implémentation de la librairie. En second lieu, nous introduisons quelques exemples d'applications concrètes des conteneurs CL_ARRAY ainsi que la classe OCL_ARRAY_VIEW. En dernier, nous advenons à l'évaluation de nos propositions sémantiques et algorithmiques.

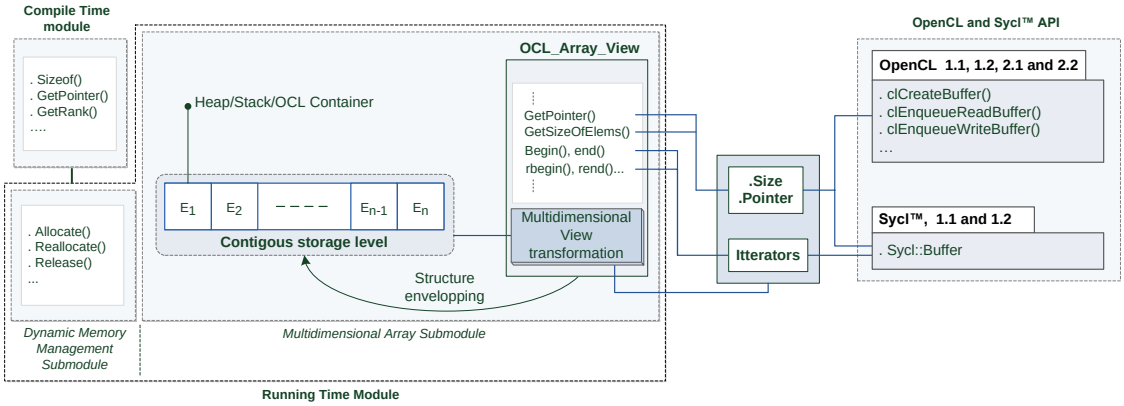
6.2 Implémentation de la librairie CL_ARRAY

La syntaxe ainsi que la sémantique prévues pour CL_ARRAY imposent une conception spécifique, cette section est prévue pour présenter notre approche pour conceptualiser les principes énoncés par le chapitre 5, ainsi que les techniques majeures qui ont permis leur mise en œuvre.
Notre implémentation est basée sur une architecture modulaire composéee de trois modules (CTM,

RTM et CL_ARRAY), chacun des modules est destiné à fournir un ensemble d'opérations sémantiques d'une même nature.

Pour simplifier l'utilisation et l'intégration de cette bibliothèque dans les programmes existants, la bibliothèque a été conçue pour être exploitable à partir d'un seul fichier d'en-tête (CL_CONTAINER.H). Pour assurer sa portabilité, elle utilise uniquement des fonctions et des méthodes faisant partie de la norme C ++ 11 [42, 85, 88] et les APIs de la norme Opencl 1.1 [67].

Figure 6.1 – Architecture de la librairie CL_ARRAY



Comme illustré dans la figure 6.1, chaque module CTM, RTM ou CL_ARRAY est implémenté dans un espace de noms séparé. Fondamentalement, Les conteneurs CL_ARRAY : :HEAP , CL_ARRAY : :STACK et CL_ARRAY : :OCL aussi bien que la structure objet OCL_ARRAY_VIEW proposée comme interface unique entre les conteneurs de la bibliothèque CL_ARRAY Sycl et les API OpenCL sont implémentés dans le sous-module Multidimensionnel (MA) de RTM, positionnant ainsi RTM en tant que module central fournissant les fonctionnalités proposées par la bibliothèque CL_ARRAY. En plus , le module RTM comprend également le sous-module DMM fournissant des fonctions de gestion de mémoire complète (Construction,Libération , Redimensionnement , Insertion et suppression des éléments) et cache la complexité des manipulations des types fournis par le module CTM.

6.2.1 Module de compilation "CTM"

```
template <class T,int N> struct _1d {typedef typename T type[N];} ;
template <class T> struct _1d<T,0> {typedef typename T type;} ;
template<class T,int N = 0> struct GetPODType :  _1d<T, 0> {};
template<class T,int N> struct GetPODType<T[N], 0> :  _1d<class _1d<T, N>::type , 0>{};
template<class T,int N,int I> struct GetPODType<T[N], I>: _1d<class GetPODType<T,I-1>::
    type ,0>{};
```

Listing 6.1 – Exemple 1 : Implémentation de la metafonction statique GetPODType

```
template <class T,size_t val> struct _type_integral  {typedef typename T type;static
    const int value=val;} ;
template<class T, unsigned N = 0> struct GetRank : _type_integral<size_t , 0> {} ;
template<class T> struct GetRank<T[], 0> : _type_integral<size_t , 0> {} ;
template<class T, unsigned N> struct GetRank<T[], N> : _type_integral<size_t , GetRank<T,
    N-1>::value+1>{} ;
template<class T, size_t N> struct GetRank<T[N], 0> : _type_integral<size_t , GetRank<T, N
    -1>::value+1>{} ;
template<class T, size_t I, unsigned N> struct GetRank<T[I], N> : _type_integral<size_t ,
    GetRank<T, I-1>::value+1>{} ;
```

Listing 6.2 – Exemple 2 :Implémentation de la metafonction statique GetRank

```
template <size_t dim,class P,class T> struct _Static_1dptr
{
    explicit Update1dptr(typename GetPODType<T,1>::type & POD, typename P & _1dptr )
    {       _Static_1dptr<dim-1,P,typename GetPODType<T, 1>::type>temp(POD[0],_1dptr) ;}
        ;} ;
template <class T,class P> struct Update1dptr<2, P,T>
{
    explicit Update1dptr(typename GetPODType<T,1>::type & POD,typename P & _1dptr )
    {  _1dptr = POD; } ; } ;
```

Listing 6.3 – Exemple 3 : Implémentation de la metafonction statique Update1dptr

Ce module, Compile Time Module (CTM) englobe les fonctions sémantiques statiques pour la manipulation des types prévues pour une exécution au compile Time. L'implémentation des ces fonctions statiques s'interprète en C++ par un ensemble de classes template qui prennent pour paramètre template des types génériques [85, 88]. L'implémentation de chaque métafonction a nécessité des traitements récursifs par spécialisation partielle et/ou totale. Les listings 6.1 , 6.2 et 6.3 présentent respectivement quelques prototypes de notre implémentation des trois métafonctions GetPODType , GetRank et Update1dptr.

6.2.2 Module d'exécution "RTM"

Ce module, Running Time Module (RTM) est destiné à fournir les structures dynamiques nécessaires à la génération et à la manipulation des collections de données multidimensionnelles. Sur le plan conceptuel et pour une meilleure rationalisation des concepts et favoriser la réutilisation des structures dynamiques, nous avons adopté une approche hiérarchique composée de deux niveaux :

6.2.2.1 Sous-module "DMM"

Le sous-module Dynamic Memory Management (DMM) est destiné à offrir les fonctionnalités liées à la gestion de la mémoire, et considérant que le Standard OpenCL est en plein essor et surtout que de nouvelles techniques pour les allocations de la mémoire sont proposées d'une version à une autre. Le DMM est conçu donc selon le patron Strategy [33], l'idée de principe est d'encapsuler les politiques de manière qu'elles soient interchangeables. Ainsi les politiques pour la gestion mémoire peuvent changer indépendamment de structures multidimensionnelles qui s'en servent.
Comme indiqué dans la section 5.2, trois stratégies sont en œuvre , elles constituent un ensemble initial de trois politiques Stackl HeaplOCl orthogonales les unes aux autres. L'avantage de cette conception est qu'elle permet l'intégration de nouvelles politiques mémoires, sans apporter de changements fondamentaux aux structures de données multidimensionnelles. Le Design par stratégie prévu pour ce module, avec les contraintes de généricité, ont nécessité une mise en œuvre à base de politiques template [10, 12, 41, 49], ce principe associe formellement à chaque stratégie d'allocation une classe Template Politique [10, 12].

Les classes Template politiques sont des "fonctor" c++ récursifs [85] métaprogrammés que nous présentons au compilateur comme des types, de cette manière, le compilateur procédera à la réécriture des portions de codes correspondantes aux politiques, en fonction de la sémantique proposée par le tableau 5.VI.

6.2.2.2 Sous-module "MA"

Le Linsting 6.4 , présente une implémentation simplifiée de la classe STACK. Le sous-module "Multidimensional Array" (MA) est destiné à fournir les enveloppes de haut niveau pour sécuriser et simplifier la gestion et la manipulation des conteneurs de données multidimensionnelles, il est conçu spécifiquement pour fournir aux utilisateurs, une mise en œuvre d'un SmartPointer pour chaque conteneur de données (CL_ARRAY : :STACK,CL_ARRAY : :HEAP à typage statique ou dynamique , et CL_ARRAY : :OCL).
Chaque SmartPointer se caractérise par une implémentation sécurisée des méthodes d'accès, ainsi que les encapsulations des fonctions proposées par le DMM ;et enfin une implémentation de la classe OCL_ARRAY_VIEW et des structures de contrôle (CL_FREE, CL_RESIEZE, CL_EXTENT , CL_INSERT , CL_DELETE, CL_MAPBUUFFER)

```
template <typename T> struct STACK
{
    static const size_t dim = GetRank<T, 0>::value;
    typedef typename GetPODType<T,dim>::type* _1d_Ptr_Type;
    T POD;
    _1d_Ptr_Type _1dptr;
    _STACK()
    {
        _Static_1dptr<dim, _1d_Ptr_Type, T> temp(POD[0], _1dptr);
    };
    typename GetPODType<T,1>::type& operator [](int index)
    {
        return POD[index];
    } ;
    .........
};
```

Listing 6.4 – Exemple 4 : implémentation d'un prototype simplifié du conteneur STACK

6.2.3 Interface utilisateur "CL_ARRAY"

Pour des raisons liées à la sécurité et de commodité d'utilisation, comme illustré dans le listing 6.5, on définit une interface qui présente les structures de contrôle et les structures de données, le but est donc de restreindre les manipulations des utilisateurs aux seules structures et méthodes publiées par l'interface CL_ARRAY.

```
1  namespace CL_ARRAY
2  {
3      template <typename MultidimensionalDataType> struct STACK
4      {
5          typename typedef RawArray::_RTM::_MA::_CL_Array<MultidimensionalDataType,RawArray
               ::_RTM::_MA::Strategy::_Stack> type;
6      };
7      template <typename MultidimensionalDataType,size_t alignment=0> struct HEAP
8      {
9          typename typedef RawArray::_RTM::_MA::_CL_Array<MultidimensionalDataType,RawArray
               ::_RTM::_MA::Strategy::_HEAP,false,alignment> build;
10     };
11     template <typename MultidimensionalDataType> struct OCL
12     {
13         typename typedef     template <typename BaseDataType,size_t N> struct
               OCL_ARRAY_VIEW
14     {
15         typename typedef ArrView::_OCL_VIEW<N, BaseDataType> type;
16     };
17     .........
18 };
```

Listing 6.5 – Implémentation du module ARRAY

6.3 Evaluation des alogirthmes proposés

```
S {double Att0 ; int Att1;  float16 Att2}
S<-S9[D0][D1]...[D8];
// measurement of build
build Container  type of S9
count=0;
// measurement access to elements
for i0 = 1:D0
 ...
  for i8 = 1:D8
        S[D0,D1,..,D8].att0<-count
        S[D0,D1,..,D8].att0<-count/2
        S[D0,D1,..,D8].att2[0]<-count/3
        ...
        S[D0,D1,..,D8].att2[15]<-count/17
        count<-count+1
// measurement of free
free(C)
```

Listing 6.6 – Programmes pour l'évaluation des temps construction, d'accès et de libération mémoire pour la structure S9

```
S<- double[a2][a1][a0]
RS<- double[b2][b1][b0]
// measurement of build
create Container C type of S
// measurement of resize
resize C with RS dimensions
// measurement of free
free(C)
```

Listing 6.7 – Programmes utilisés pour l'évaluation des temps de construction, de redimensionnement et libération mémoire

Cette section est dédiée à l'évaluation des algorithmes proposés pour la construction, le redimensionnement et la libération de la mémoire que nous comparons avec trois librairies de références Blitz++, std : :vector et boost::multiarray. Les tableaux multidimensionels du std : :vector sont obtenus avec la formulation vector of vector ..of vector ($std :: vector < std :: vector < ... >> (....)$). Vu la capacité des conteneurs CL_ARRAY à opérer dans des écosystèmes spécialisés tel que Opencl et d'autres ecosystemes (multipurpose) basés sur un parallèlisme à mémoire partagée distribuée, nous basons notre évaluation sur deux benchmarks exécutés sur le même ordinateur équipé avec 32 GO de RAM, un processeur i7 3770, une carte GPU AMD Radeon 7970 mais avec deux systémes d'exploitation et deux différents compilateurs (Microsof Visual Studio 2011 MSV sous windows et G++ 4.8) :

1. Système d'exploitation Microsoft Windows 7 64-bit exécutant le compilateur Microsoft Visual Studio 2012 (MVS) et la distribution Windows d'AMD OpenCL SDK version 3.0

2. Système d'exploitation Linux Debian 14.0.4, avec le compilateur G++ version 4.9 et la distribution linux d'AMD OpenCL SDK version 3.0.

3. Pour le compilateur MVS sous windows, les statistiques mémoires sont obtenues par l'utilisation de l'API Windows GetProcessMemoryInfo [1].

4. Pour le compilateur Linux G++, les statistiques sont obtenues en utilisant l'API sysinfo [2] et en utilisant l'outil linux $/usr/bin/time$ [3] pour la mesure de la consommation mémoire maximale.

6.3.1 Évaluation de l'impacte de la Dimensionnalité

Bien qu'en général les applications qui requièrent un grand nombre de dimensions ne sont pas très communes, il n'en demeure pas moins que quelques applications spécifiques requièrent une forte dimensionnalité à l'instar du business intelligence (BI) [57]. En effet , les applications BI qui utilisent le processus (MOLAP) stockent leurs données dans un tableau multidimensionnel à forte dimensionnalité (>>8) [50, 57] plutôt qu'une base de données relationnelle [50].

Pour les besoins de cette évaluation, nous proposons un tableau de structure multidimensionnel (Array Of Structure) AoS non-alignée (taille des éléments qui n'est pas en puissance de 2) à l'instar de la structure proposée avec des attributs {int, double, float16} qui peut provoquer en C++ de nombreux problèmes de performance. Comme le montre le tableau suivant , nous proposons un intervalle de test de neuf à seize dimensions pour le AoS. Nous exécutons le pseudo-code donné dans 6.6 pour chaque structure {S9,S10...,S16}.
Pour le compilateur Visual studio, à contrario des conteneurs CL_ARRAY et boost::multiarray qui

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/ms683219(v=vs.85).aspx
[2] http://man7.org/linux/man-pages/man2/sysinfo.2.html
[3] http://man7.org/linux/man-pages/man1/time.1.html

n'imposent aucune restriction sur le nombre de dimensions, ceux de la STL sont limités ; lors de nos tests nous n'avons pas pu dépasser huit dimensions. Par conséquent, les conteneurs std : :vector sont exclus de cette évaluation du grand nombre de dimensionnalité sous Visual Studio.

Tableau 6.I – Definition des Types

S est une Structure définie comme suit : struct S{double Att0 ; int Att1 ; float16 Att2}

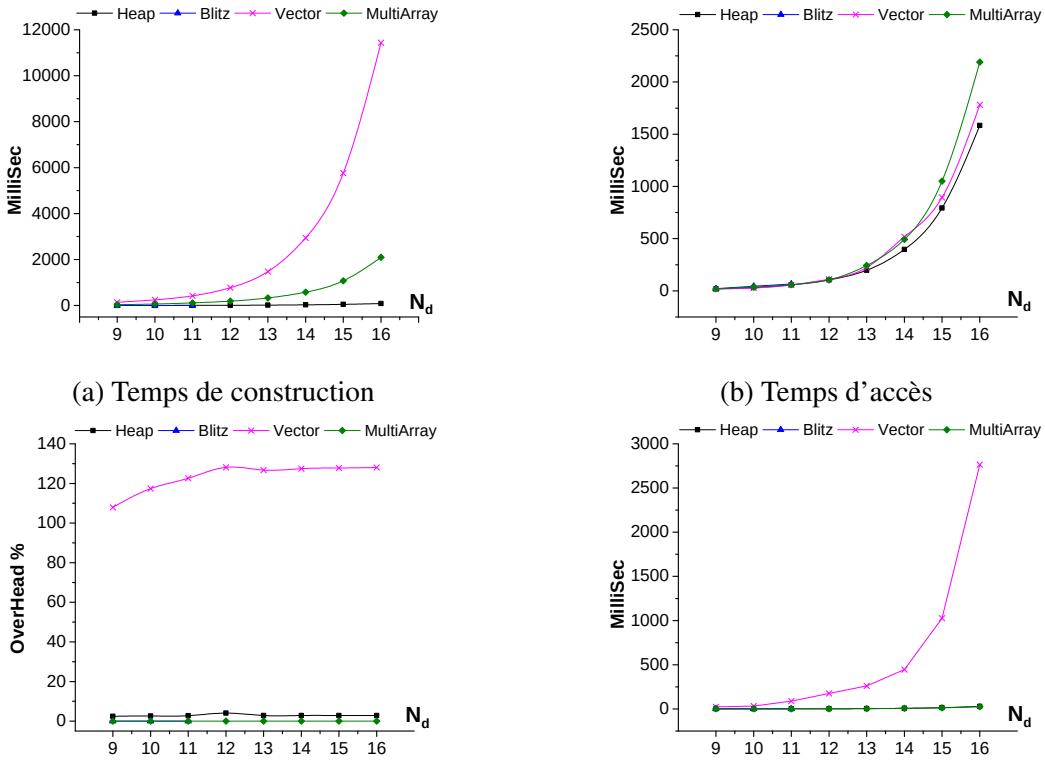
N_d	*Type Definition*	N_d	*Types*
9	S9[2][2][2][2][20][20][19][2][3]	13	S13[2][2][2][2][2][2][2][2][20][20][19][2][3]
10	S10[2][2][2][2][2][20][20][19][2][3]	14	S14[2][2] [2][2][2][2][2][2][2][20][20][19][2][3]
11	S11[2][2][2][2][2][2][20][20][19][2][3]	15	S15[2][2][2][2][2][2][2][2][2][2][20][20][19][2][3]
12	S12[2][2][2][2][2][2][2][20][20][19][2][3]	16	S16[2][2][2][2] [2][2][2][2][2][2][2][20][20][19][2][3]

6.3.1.1 Évaluation des temps de construction

Les figures 6.2a et 6.3a présentent les temps requis pour la construction de chacune des huit structures (de neuf à seize dimensions) pour les conteneurs de boost::multiarray, CL_ARRAY::HEAP, Blitz++ et std::vector (seulement pour G++) .

Les temps requis pour la construction des conteneurs avec les deux compilateurs sont égaux en utilisant Blitz++ et sont sensiblement inférieurs à ceux obtenus en utilisant CL_ARRAY : :HEAP , tandis que std::vector et boost::multiarray (MVS) ont montré les temps de construction les plus élevés. Sous MVS, la dimensionnalité (augmentation dans le nombre des dimensions) affecte considérablement les performances de boost::multiarray . Cette dégradation est proportionnelle au nombre de dimensions, elle atteint son maximum à seize dimensions (temps de construction sont 30 fois supérieurs que ceux requis en utilisant le conteneur de CL_ARRAY::HEAP).

Pour le compilateur G++ comparativement à CL_ARRAY::HEAP , les temps de construction pour std::vector sont jusqu'à 134-fois supérieurs tandis que les conteneurs de boost : :multiarray sont de 22 à 27-fois plus élevés.



(a) Temps de construction

(b) Temps d'accès

(c) Surcharge mémoire en %

(d) Temps de liberation mémoire

Figure 6.2 – Evaluation de l'impacte de la forte dimensionnalité - Linux ubuntu 64 bits avec GCC 4.9

6.3.1.2 Evaluation des temps d'accès et de la consommation mémoire

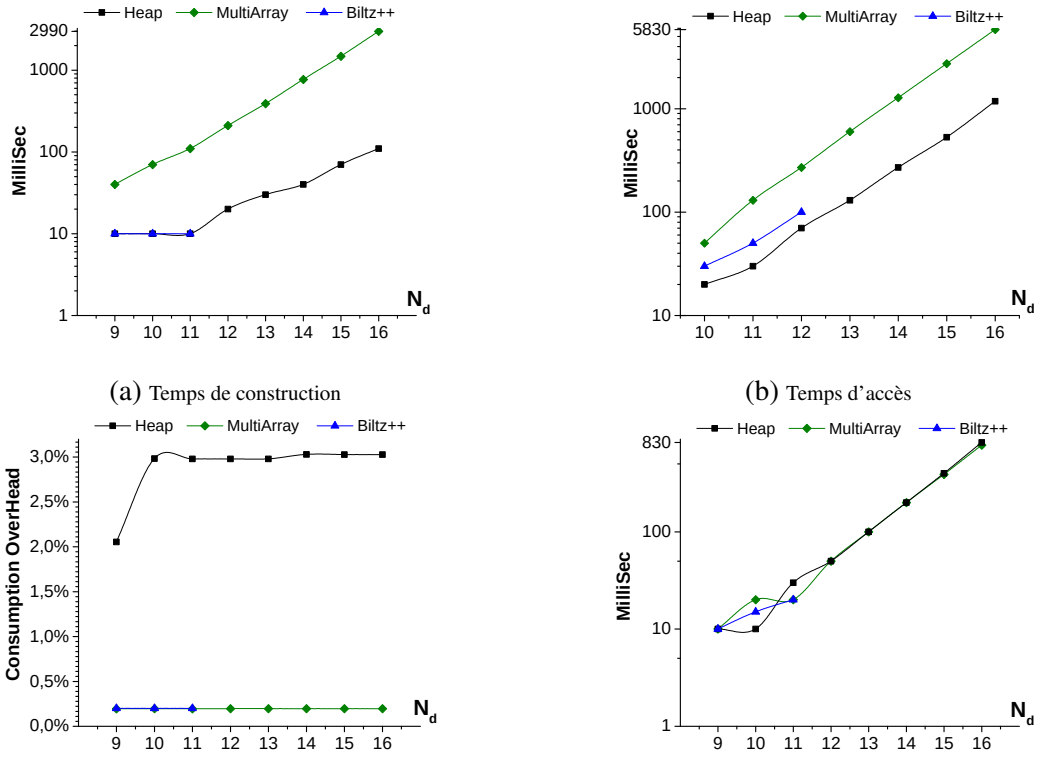
Les figures 6.2b et 6.3b représentent les temps d'accés pour les deux compilateurs. Les temps observés pour les conteneurs de CL_ARRAY::HEAP sont les plus bas, seuls les temps pour les conteneurs de std::vector pour les trois première dimensions sous le compilateur G++ étaient légèrement inférieurs. Comme le montrent les figures 6.2c et 6.3c, ces améliorations sont expliqués par le comportement de l'allocateur mémoire par défaut de std : :vector, qui tend à améliorer l'alignement mémoire par l'allocation d'une mémoire alignée en puissance de 2 mais au détriment d'une flagrante surconsommation mémoire .En effet, le résultat constaté est que std : :vector consomme en moyenne deux fois plus de mémoire que requis. Alors que Blitz++, boost : :multiarra

et CL_ARRAY : :HEAP présentent respectivement 0.5%, 2% et 3% de surconsommation mémoire (overhead).

Les conteneurs boost::multiarray ont montré les temps d'accès les plus élevés ,et ce, pour les deux compilateurs (MSV et G++), comparativement à CL_ARRAY::HEAP, ils sont de 5 à 38% plus élevés avec G++ et de 2 à 5 fois plus élevés avec le compilateur MVS.

Les temps d'accès des conteneurs de std::vector et Blitz++ occupent une position médiane et sont respectivement de 5 à 12% et de 40 à 60% plus élevés que ceux obtenus en utilisant CL_ARRAY::HEAP.

6.3.1.3 Évaluation des temps de libération :



(a) Temps de construction

(b) Temps d'accès

(c) Surcharge mémoire en %

(d) Temps de liberation mémoire

Figure 6.3 – Evaluation de l'impacte de la forte dimensionnalité - Windows 7 64 bits avec Microsoft Visual Studio 12.0

Les figures 6.2d et 6.3d montrent les temps requis pour la libération de chaque conteneur. Théoriquement, l'algorithme utilisé pour la libération mémoire des conteneurs CL_ARRAY::HEAP requière N_d opérations de libération mémoire (où N_d représente la dimensionnalité du conteneur). Nous constatons que tous les conteneurs analysés (boost::multiarray, CL_ARRAY::HEAP et Blitz++) à l'exception de std::vector, requièrent des temps de libération similaires, cette similarité s'explique en se basant sur les figures 6.2c et 6.3c , par la taille mémoire du niveau d'accès qui occupe au plus 3% de l'espace mémoire en comparaison avec le niveau de stockage.
Par conséquent, les temps requis pour libérer la mémoire occupée par le niveau d'accès est négligeable comparativement au niveau de stockage.

6.3.2 Évaluation de la capacité de contenance des données

Pour les besoins de cette évaluation, comme le montre le pseud-code du listing 6.7, toutes les structures sont de type double et sont de la même dimensionnalité égale à trois (tableaux tridimensionnels). Nous fixons le niveau d'accès des structure à [2000][200] et nous varions les capacité de stockage avec les valeurs respectives suivantes {500,1000,2000,4000},le scénario proposé pour cette évaluation s'exprime comme suit :

1. $\begin{cases} S1 \triangleq double[2000][200][500], S2 \triangleq double[2000][200][1000] \\ S3 \triangleq double[2000][200][2000], S4 \triangleq double[2000][200][4000] \\ RS1 = S2, RS2 = S3, RS3 = S4 \end{cases}$
2. RS1,RS2,RS3 sont obtenus respectivement après redimensionnement de S1,S2,S3

6.3.2.1 Evaluation des temps de construction

Avec MVS comme illustré dans la figure 6.6a, le temps requis pour construire les conteneurs std::vector est trois à quatre-fois plus élevé que celui requis par boost::multiarray. Cette surconsommation de temps s'explique par les abstractions objets complexes utilisées par std::vector et boost::multiarray.

La construction des structures CL_ARRAY::HEAP comme pour celles de Blitz++ sous l'OS Windows, requière l'intervention de l'allocateur système. En l'absence d'opérations d'accès, ces allocateurs vont simplement allouer de la mémoire virtuelle.

Avec G++, comme illustré dans la figure 6.4a, les temps d'accès sont sensiblement les mêmes pour CL_ARRAY::HEAP et Blitz++ , alors que les temps de construction pour boost::multiarray et std::vector sont respectivement en moyenne de 35 à 56% supérieurs à ceux observés en utilisant CL_ARRAY : :HEAP.

6.3.2.2 Evaluation des temps de libération

Comme le montre la figure 6.6c, les trois librairies (boost::multiarray, CL_ARRAY::HEAP, et Blitz++), requièrent sous le compilateur MVS des temps similaires pour libérer la mémoire d'un conteneur. Par contre le temps requis pour libérer la mémoire utilisée par les conteneurs std::vector est approximativement 90 fois supérieur à celui de CL_ARRAY : :HEAP.

Sous G++, les temps de libération sont montrés dans la figure6.4c. Les temps requis par Blitz++ sont les plus bas. Comparativement, CL_ARRAY::HEAP a requis 6 à 7 fois plus de temps, tandis que boost::multiarray et std::vector sont les conteneurs les plus lents à libérer avec des temps pratiquement similaires qui ont requis 17 fois plus de temps.

6.3.2.3 Evaluation des temps de redimensionnement

Comme le montre la figure 6.6b, les opérations de redimensionnement pour les conteneurs std::vector sous le compilateur MVS requièrent 9 à 14 fois plus de temps que ceux observés en utilisant les conteneurs de CL_ARRAY::HEAP. Parallèlement, les temps de redimensionnement requis par les conteneurs boost::multiarray sont plus élevés(> 20%), tandis que ceux requis par Blitz++ sont de 3 à 5% inférieurs à ceux observés en utilisant CL_ARRAY ::HEAP.

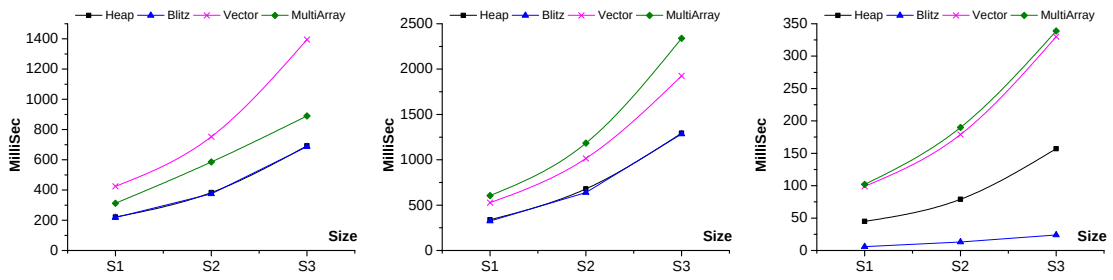
Avec le compilateur G++, comme le montre la figure 6.4b, boost::multiarray requière approximativement deux fois plus de temps en comparaison avec les conteneurs de CL_ARRAY::HEAP. Les temps de redimensionnement pour les conteneurs std::vector sont de 45 à 55% plus élévés , alors que ceux recquis par Blitzz+ sont de 0.7 à 2.5% inférieurs.

6.3.2.4 Evaluation de la consommation mémoire

Pour cette évaluation ,nous mesurons et comparons la quantité mémoire maximale consommée avant et après les opérations de redimensionnement (voir les figures 6.5a et 6.7a) .

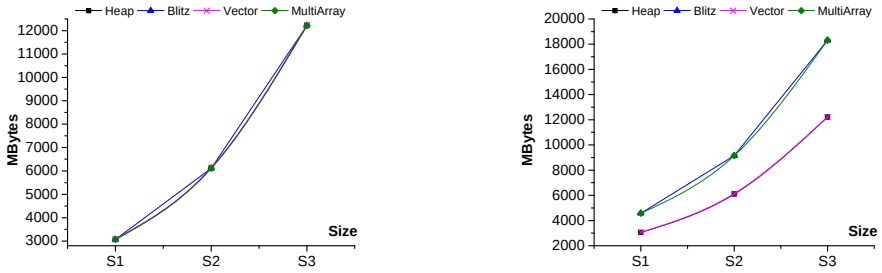
Avant l'opération de redimensionnement : les structures std::vector consomment sous Windows et sous linux 2 à 6% de mémoire en plus par rapport à CL_ARRAY::HEAP, boost::multiarray et Blitz++ . Cette consommation de mémoire additionnelle s'explique par le comportement des allocateurs mémoire qui allouent, un peu plus de mémoire que nécessaire.

Après l'opération de redimensionnement : Les figures 6.7b et 6.5b montrent que la consommation mémoire maximale sous linux est pratiquement la même que celle sous Windows (les résultats sous linux sont approximativement 1% inférieur). Le redimensionnement des conteneurs boost::multiarray requière en moyenne 60% plus de mémoire et Blitz++ a requis de 12 à 30% plus de mémoire en comparaison avec la quantité mémoire consommée par le conteneur CL_ARRAY::HEAP.



(a) Temps de construction (b) Temps de redimensionnement (c) Temps de liberation mémoire

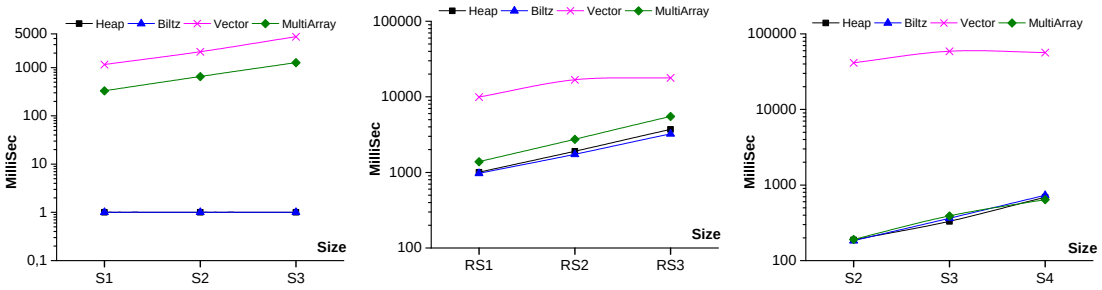
Figure 6.4 – Évaluation de la capacité des conteneurs à gérer une très grande volumétrie de données - Linux ubuntu avec G++ 4.9



(a) Pic de Consommation mémoire avant redimensionnement en MB

(b) Pic de Consommation mémoire après redimensionnement en MB

Figure 6.5 – Évaluation de la Consommation mémoire - Linux ubuntu avec G++ 4.9

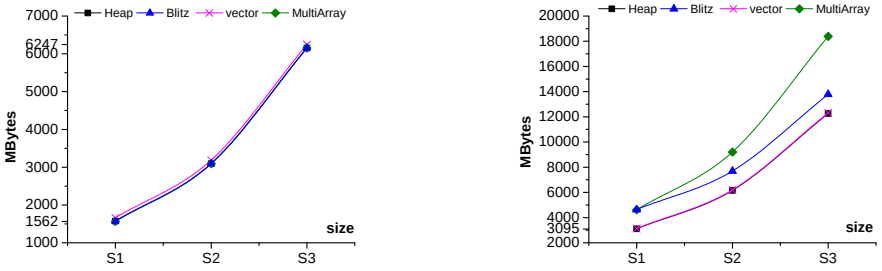


(a) Temps de construction

(b) Temps de redimensionnement mémoire

(c) Temps de libération mémoire

Figure 6.6 – Évaluation de la capacité des conteneurs à gérer une très grande volumétrie de données - Windows 7 64 bits avec Microsoft Visual Studio 12.0



(a) Pic de Consommation mémoire avant redimensionnement en MB

(b) Pic de Consommation mémoire après redimensionnement en MB

Figure 6.7 – Évaluation de la Consommation mémoire - Windows 7 64 bits avec Microsoft Visual Studio 12.0

6.4 Application pour la programmation générique

```
using namespace CL_ARRAY;using namespace std;
#define N ...
STACK<double[N][N]> A0;
HEAP<double[N][N]> A1;// HEAP Unrealisable two-dimensionality
OCL<double[N][N]> A2(CL_MapBuffer(..));// OCL two-dimensionality
HEAP<double,2> A3(N,N);// Heap realisable two-dimensionality
double A4[N][N]; // POD two-dimensionality  Array
MatMultPOD(A0,A1,A3);MatMultPOD(A0,A1,A3);...; //Any combination of arrays works
```

Listing 6.8 – Exemple UAP

6.5 Application interfaçage avec la librairie standard C++(STL)

```
CL_ARRAY::HEAP<double[10][10][10][10][2][2]> A;double n1=...;
OCL_ARRAY_VIEW<double,6> MyView (10,10,10,10,2,2);MyView=A;
//Fill interval [5][0][0][0][0][0] to [5][10][10][10][2][2] by -1
std::fill(MyView[5].begin(), MyView[5].end(), -1);
//Find the element n1 in the interval [10][0][0][0][0][0]-[10][0][10][10][2][2]
auto result1 = std::find(MyView[10][0].begin(), MyView[10][0].end(), n1);
//Find the element n1 in the all structure
auto result2 = std::find(MyView.begin(), MyView.end(), n1);
```

Listing 6.9 – Exemple d'illustration des conteneurs HEAP et OCL_ARRAY_VIEW pour l'interfaçage avec la STL

Le listing 6.9 illustre un exemple d'interaction entre les conteneurs CL_ARRAY et les algorithmes STL utilisant l'interface ittérateur mutlidiementionel offert par OCL_ARRAY_VIEW. Comme indiqué dans la liste 6.8, tous les conteneurs CL_ARRAY (STACK, HEAP et OCL) utilisent une méthode d'accès uniforme comme des tableaux multidimensionnels POD, simplifiant ainsi l'écriture de programmes génériques en évitant la nécessité d'écrire différentes versions comme mentionné dans 5.1.

6.6 Application MAP pattern

6.6.1 Scenario

```
using namespace CL_ARRAY ;
# define StructType double[a0][a1][a2]
//OpenCL API invocation.
...
//Begin OCL_VIEW_SECTION.
OCL< StructType > Input(MAPBuffer);
OCL< StructType > Output(MAPBuffer);
OCL_ARRAY_VIEW<double,2> InputView (a0, a1*a2);
OCL_ARRAY_VIEW<double,2> OutputView(a0, a1*a2);
InputView= Input; OutputView=Output;
/*Update Input*/
for(size_t i=0;i<a0;i++)
for(size_t j=0;i<a1;j++)
for(size_t k=0;k<a2;k++)
{
Input[i][j][k] = ...;Output[i][j][k] = ...;
}
/*begin data transfer in the GPU memory*/
for(size_t i=0;i<a0;i++)
{
clEnqueueWriteBuffer(..,InputView[i].GetSizeOfElems(),InputView[i].GetPtr(),..);
clEnqueueWriteBuffer(..,OutputView[i].GetSizeOfElems(),InputView[i].GetPtr(),..);
//1 dimensional Ndrang= InputView[i].GetElemSize().
clEnqueueNDRangeKernel(...,InputView[i].GetSizeOfElems(),..);
clEnqueueReadBuffer(..,OutputView[i].GetSizeOfElems(),OutputView[i].GetPtr(),..);
};
for(size_t i=0;i<a0;i++)
for(size_t j=0;i<a1;j++)
for(size_t k=0;k<a2;k++)
{
//Display the results after MAP parallel processing
std::cout<<Output[i][j][k]<<std::endl;
}
```

Listing 6.10 – Exemple d'illustration des conteneurs HEAP et OCL_ARRAY_VIEW pour l'interfaçage avec les APIs OpenCL

Map est un paradigme largement utilisé dans la programmation parallèle, ce modèle stipule qu'une même tâche est appliquée à tous les éléments ou à une partie d'un conteneur de données. Les opérations Map sont utilisées pour résoudre des problèmes qui peuvent être décomposés en sous-tâches indépendantes, tel que l'addition matricielle où chaque sous-tâche ne nécessite aucune communication ou synchronisation. Le listing 6.10 présente un exemple d'un scenario trivial utilisant un pattern MAP pour une application d'addition de deux matrices à trois dimensions en utilisant OpenCL. L'implémentation présentée additionne deux conteneurs CL_ARRAY à trois dimensions ($Output$ = $Input + Output$) à trois dimension [A0][A1][A2] de type double, les conteneurs ($Output et Input$) sont de type OCL. Afin d'améliorer la portabilité du programme, le processus a été distribué en A0 tâche.

Ce scénario requière deux conteneurs OCL CL_ARRAY : :OCL et deux vues à deux dimensions [A0][A1*A2] de type double pour les structures (Input) et de sortie (Output) , un NDrange à une seule dimension.

Ainsi , les deux vues à deux dimensions requièrent une transformation multidimensionnelle 2d $[a0][a1 * a2]$, permettant de produire le partitionnement suivante :

$\underbrace{[a0]}_{\text{Number of kernel executions}}$; chaque kernel exécute $\underbrace{[a1 * a2]}_{\text{1d Ndrange with WG=a1*a2}}$ éléments.

Cette transformation multidimensionnelle (de trois à deux dimensions) est fournie pour les deux conteneurs OCL (Input et Output) par OCL_ARRAY_VIEW .

6.6.2 Évaluation du tau de transfert des données

Dans cette évaluation, nous comparons les capacités des conteneurs CL_ARRAY::HEAP (conçus pour opérer selon une technique de transfert par défaut) et CL_ARRAY::OCL. Cette évaluation est le résultat d'exécution des programmes cités dans l'exemple du listing 6.10. Ces programmes requièrent deux structures à trois dimensions de 1 GO de taille en entrée. Le traitement est réparti en dix portions de 100 MO chacune. Comme montrée dans la table 6.II, hormis la première tentative d'opération de lecture, les structures OCL ont permis d'atteindre des vitesses de transfert maximales (Peak transfert speeds) – doublant les taux de transfert comparativement au transfert par défaut utilisé par les structures CL_ARRAY::HEAP.

	clEnqueueWriteBuffer		*clEnqueueReadBuffer*		
Copy	*CL_ARRAY::OCL*	*CL_ARRAY::HEAP*	*CL_ARRAY::OCL*	*CL_ARRAY::HEAP*	*SIZE*
1	55.3526	25.4502	42.8062	14.9987	100 MB
2	9.0935	14.5933	8.6843	14.1640	100 MB
3	8.9419	14.6741	8.4797	14.0902	100 MB
4	8.8284	15.0960	8.4416	14.4865	100 MB
5	8.8133	15.1551	8.3824	14.8145	100 MB
6	8.9301	15.0061	8.4313	14.5748	100 MB
7	8.8843	14.9498	8.4846	15.2385	100 MB
8	8.7807	15.2852	8.4130	15.0443	100 MB
9	8.8364	15.4834	8.4907	15.2218	100 MB
10	8.7527	15.5281	8.3803	14.4686	100 MB

Tableau 6.II – Mesure du temps de transfert en secondes

Pour le second scenario, les conteneurs CL_ARRAY ::HEAP ont été alignés à 128 octets [55], nous avons utilisé le SDK intel 2014 SDK release et les dispositifs : "CPU" Intel i7 et "APU" intel HD4400 APU). Pour ce présent scénario nous avons constaté un meme niveau de performances des conteneurs OCL et HEAP en appliquant la stratégie «Zero Memory Copy», donc non concerné' par les transferts PCI-Experss.

6.7 Application "Algèbre Multilinéaire" de benchmarking

```
//C conventional 2d array
double** Array2d;
Array2d=(double**)malloc(X*sizeof(double*));
for(int i=0;i< X;i++)
 Array2d[i]=(double*)malloc(Y*sizeof(double));
```

Listing 6.11 – Tableau 2D C conventionnel

```
m=p=n=size
for i = 1:m
for j = 1:p
for k = 1:n
   C[i,j,K] <- A[i,j,K]+B[i,j,K]
```

Listing 6.12 – Addition de deux tensor à trois dimensions

```
l=m=p=q=n=size
for h = 1:l
 for i = 1:m
  for j = 1:p
   for g = 1:q
     for k = 1:n
     C[h,i,j,g] <- C[h,i,j,g]+A[h,i,k]*B[k,j,g]
```

Listing 6.13 – Contraction tensorielle à trois dimensions

```
m=n=size
for i = 1:m
 for j = 1:n
   for k = 1:n
     C[i,j] <- C[i,j]+A[i,k]*B[k,j];
```

Listing 6.14 – Pseudocode contraction tensorielle à deux dimensions (Multiplication matricielle)

Dans cette section nous présentons un benchmark pour l'analyse des statistiques des défauts de caches de niveaux deux et trios pour l'évaluation de l'impacte de la localité mémoire, de l'énergie consommée et des temps de calcul dans le contexte de trios applications de l'Algèbre multilinéaire. Les applications sont choisies selon la complexité des nids de boucles et des parcours (access patern) [33, 63, 95], les applications concernent (comme montré dans le tableau 6.III) :

1. une contraction tensorielle à trois dimensions basée sur un algorithme au parcours complexe montré dans le tableau 6.III)

2. une contraction tensorielle (application de multiplication matricielle) à deux dimensions qui représente un cas d'étude largement répandu pour l'évaluation des performances des

conteneurs (montré dans 6.14 [71]) et des méthodes d'accès qui nécessitent un parcours moyennement complexe

3. une addition de deux matrices à trois dimensions basée sur un modèle d'accès en ligne majeur montré dans 6.12.

Tableau 6.III – boucles imbriquées et modèle d'accès

Application	**Imbriquation**	**Mdèle d'accés**
Addition tensorielle à tridimensionnelle	trois boucles	A, B, et C ligne-majeur
Contraction tensorielle bidimensionnelle	trois boucles	A et C ligne majeur colonne majer
Contraction tensorielle tridimensionnelle	Cinq boucles	A et C ligne-majeur B modèle d'accès complexe

Tous les tableaux dans cette évaluation sont de type double et leurs tailles par dimensions pour chaque application, sont présentées dans le tableau 6.IV.

Tableau 6.IV – Dimension des tableaux pour chaque application d'Algèbre multilinéaire

Application	**Size per dimension**	**Pas**
Addition tensorielle à tridimensionnelle	$A[N,N,N], B[N,N,N], C[N,N,N]$	$N \in [100, 200, .., 800]$
Contraction tensorielle bidimensionnelle	$A[N,N], B[N,N], C[N][N]$	$N \in [100, 200, .., 3200]$
Contraction tensorielle tridimensionnelle	$A[N,N,N,N], B[N,N,N], C[N,N,N]$	$N \in [20, 40, .., 120]$

Pour cette évaluation , nous comparons les conteneurs CL_ARRAY::HEAP aux conteneurs multidimensionnels équipés de pointeurs qui exposent une méthode d'accès standard (par opérateur subscript) pour chaque application. Nous comparons également CL_ARRAY::HEAP à trois autres

librairies de conteneurs objets métaprogrammées très répandues auprès des développeurs. Les deux premières librairies qui sont vector de la STL (std : :vector) avec la formulation vector of vector..of vector et boost::multiarray produisent toutes deux des conteneurs multidimensionnels à mémoire disparate avec une méthode d'accès standards par opérateur subscript. La troisièmes librairie est Biltz++, et est représentative des conteneurs multidimensionnels pseudo-array. Parallèlement, quelques developpeurs C ont parfrois tendance à écrire leurs prorpes implémentations de tableaux multidimentionels dynamiques avec la sémantique des pointeurs multidimentionnels qui produit des tableaux C également à mémoire disparate. Nous appelons un conteneur produit par cette méthode un « conteneur C conventionel » (Le listing 6.11 fournit une vue sur la mise en oeuvre d'un conteneur C conventionel à deux dimensions). Nous avons manuellement implémenté par cette méthode trois conteneurs à deux, à trois et à quatre dimensions (Array2d, Array3d et Array4d respectivement) et avons utilisé ces conteneurs dans cette évaluation. Pour chaque application, en utilisant les conteneurs C conventionnels, les conteneurs de boost : :multiarray et les conteneuurs std : :vector produits par la formulation vector of vector of vector, nous présentons dans quatre figures séparées. Pour présenter respectivement les statistiques des défaut de cache de niveau deux et trois, l'énergie consommée en joules et les temps d'exécution en pourcentage. Ces pourcentages sont obtenus par l'application de la formule suivante : $HEAP_{time} - OtherContainer_{time}/HEAP_{time}$. Ainsi, les pourcentages positifs indiquent une amélioration tandis que les pourcentages négatifs indiquent une dégradation par rapport aux utilisations CL_ARRAY : :HEAP. Les statistiques sont obtenues après l'éxécution de chacune des applications et pour chaque type de conteneur sur un ordinateur équipé de 32 GO de mémoire, d'un processeur I7 3770, du système d'exploitation linux Debian 14.0.4 et de la version 4.9 du compilateur GCC/G++ avec l'option de compilation O3 [91] (cette option est recommandée , elle inclut une large variété d'algorithmes d'optimisation et l'auto vectorisation). Pour la collecte des statistiques, nous avons utilisé la version 2.8 d'Intel Performance Counter Monitor. Les applications choisies pour l'évaluation nous permettent de statuer sur :

1. Les performances des librairies 'vector de la STL' et 'multiarray de boost' qui sont très dégradées en terme de temps d'éxécution et d'énergie consommée comme exprimé dans les graphes ci-dessous

- Comme le montrent les figures 6.8c et 6.8d, pour la contraction tensorielle à trois dimensions, le temps d'exécution est de cinq à onze fois supérieur lors de l'utilisation des conteneurs std : :vector et trois à huit fois supérieur lors de l'utilisation des conteneur boost : : multiarray par rapport au temps d'exécution observés pour les conteneurs CL_ARRAY containers.
- Comme le montrent les figures 6.9c et 6.9d, pour la multiplication matricielle à deux dimensions, l'écart entre librairies est moindre par rapport à l'application contraction tensorielle à trois dimensions mais les dégradations restent importantes : l'utilisation des conteneurs vector a mené à des temps d'exécution de deux à huit fois supérieurs à ceux des conteneur multiarray de boost qui sont à leur tour de deux à quatre fois supérieurs a ceux constatés pour les conteneurs CL_ARRAY.
- Comme le montrent les figures 6.10c et 6.10d pour l'addition matricielle à trois dimensions, l'écart est assez conséquent : l'utilisation des conteneurs vector a mené à des temps d'exécution de quatre à quatorze fois supérieurs à ceux des conteneurs multiarray de boost qui sont à leur tour de trois à treize fois supérieurs à ceux constatés pour les conteneurs CL_ARRAY.

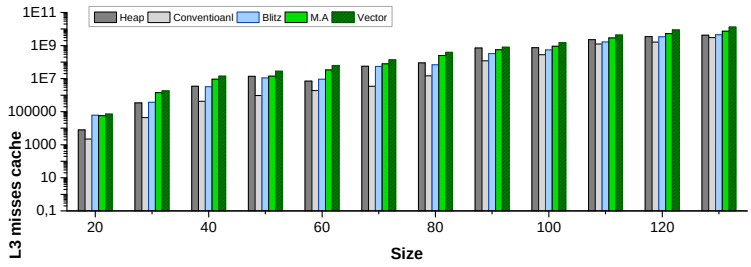
2. L'impact de la mauvaise localité de mémoires des conteneurs qui sont obtenues par la méthode conventionnelle. Nous avons dans ce cas observé ce qui suit :

   - Comme le montrent les figures 6.8b et 6.8a pour la contraction tensorielle à trois dimensions, l'utilisation des conteneurs CL_ARRAY a permis une nette réduction du nombre de défauts de caches L2 et L3 de 20 à 60% par rapport aux conteneurs conventionnels, avec comme résultat des temps d'exécutions réduits de 15 à 75%
   - Comme le montrent les figures 6.9b et 6.9a pour l'application multiplication matrcielle à deux dimensions, l'utilisation des conteneurs CL_ARRAY a permis une nette réduction du nombre de défauts de caches L2 et L3 de 20 à 200% par rapport aux conteneurs conventionnels avec comme résultat des temps d'exécutions réduits de 4 à 400%
   - Comme le montrent les figures 6.10b et 6.10a pour l'application addition tensorielle à

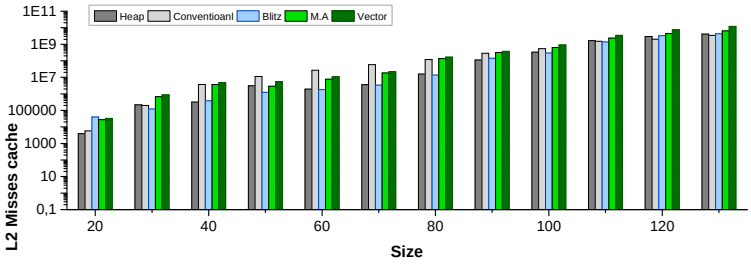
trois dimensions, nous constatons de nouveau que l'écart est assez conséquent, l'utilisation des conteneurs CL_ARRAY a permis une nette réduction du nombre de défauts de caches L2 et L3 de 5 à 15% par rapport aux conteneurs conventionnels avec comme résultat des temps d'exécutions réduits de 5 à 8%

3. les performances atteintes par les conteneurs de CL_ARRAY::HEAP et de Blitz++ pour les trois applications et qui sont similaires avec un léger avantage pour CL_ARRAY.

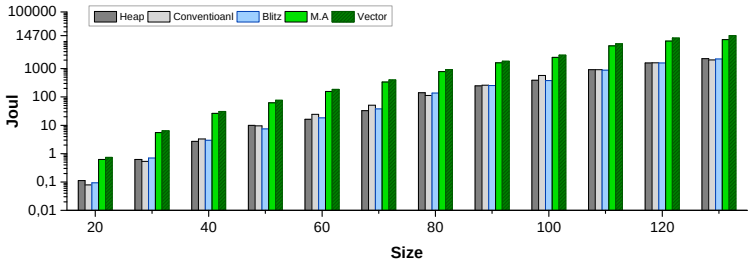
Les dégradations constatées pour les deux librairies vector et multiarray révèlent toute la difficulté des compilateurs d'opérer à des optimisations efficaces en présence des librairies objets, cette difficulté s'interprète par une dégradation des performances qui s'accroit proportionnellement avec la complexité des algorithmes. Parallèlement , les compilateurs sont bien plus efficace en présence de conteneurs produits par la méthode conventionnelle ; par conséquent, nous constatons de meilleures performances lors de l'utilisation des conteneurs CL_ARRAY par rapport aux deux librairies objets Vector et MultiArray, par ailleur, ces conteneurs présentent des performances faibles par rapport aux conteneurs Biltz++. Bien que la méthode C conventionnelle aussi bien que les méthodes proposées dans la littérature produisent des conteneurs multidimensionnels équipés d'une arborescence de pointeurs multidimensionnels et une méthode d'accès aux structures de données par opérateur Subscript ; toutefois et à l'opposé de la méthode conventionnelle, la méthode proposée pour la création des conteneurs CL_ARRAY produit des conteneurs résidant dans des zones mémoires contigües ; cette nouvelle configuration mémoire a permis une forte réduction du nombre des défaut de caches L2 et L3 et l'obtention par conséquence, de performances supérieures, d'importantes économies d'énergies et d'égaler voire de surpasser les performances de la méthode par calcul d'indexe utilisé par Blitz++ et ce, pour les trois applications proposées d'algèbre multilinéaire. Sur la base de nos résultats, CL_ARRAY : :HEAP et CL_ARRAY : :OCL proposent une méthode d'accès par opérateur subscript qui à la fois adhère au standards et garantit une interopérabilité la plus large possible avec la majorité des librairies séquentielles ou parallèles. CL_ARRAY offre un contexte précis pour représenter la dimensionnalité sans aucune restriction (Biltz++ limite les conteneurs Array à onze dimensions) et permet de meilleures performances qui s'affirment davantage en présences d'algorithmes plus complexes par rapport aux conteneurs C/C++ conventionnels.



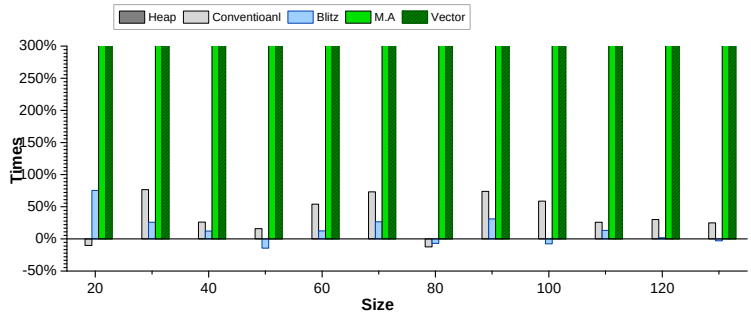
(a) Défauts de caches de niveau trois



(b) Défauts de caches de niveau deux

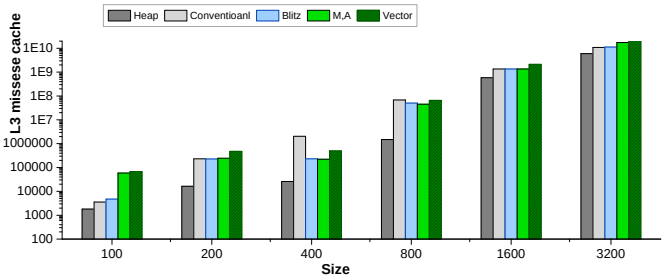


(c) Consomation d'énergie Vector, Blitz++, multiarray et CL_ARRAY::HEAP en joules

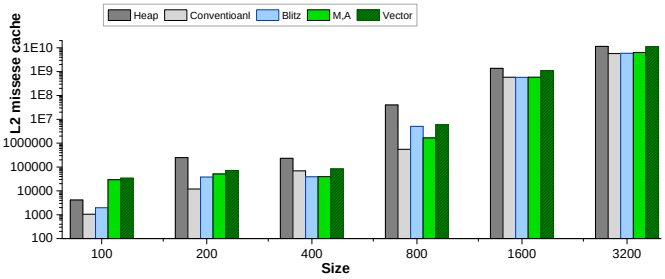


(d) *Temps d'exécution Vector, Blitz++, et multiarray en percentage comparé à CL_array::HEAP*

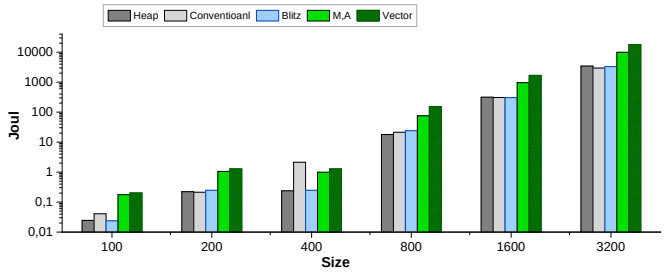
Figure 6.8 – Statistiques obtenues pour l'application contraction tensorielle à trois dimensions



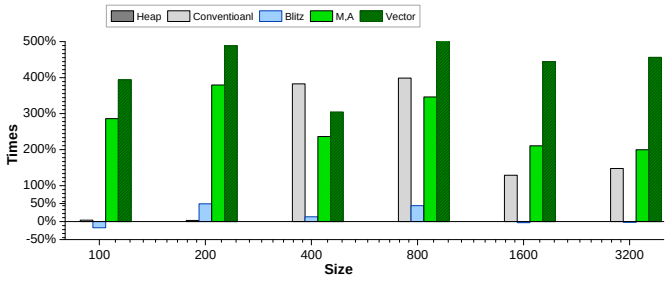
(a) Défauts de caches de niveau trois



(b) Défauts de caches de niveau deux

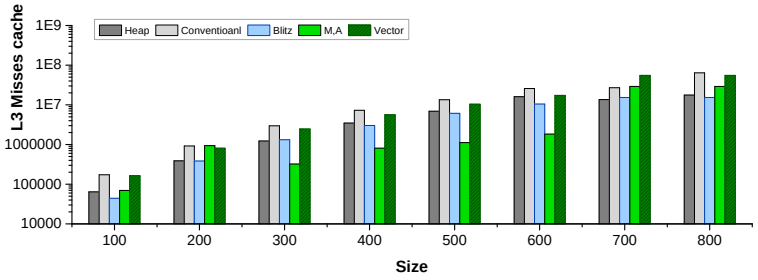


(c) Consomation d'énergie Vector, Blitz++, multiarray et CL_ARRAY::HEAP en joules

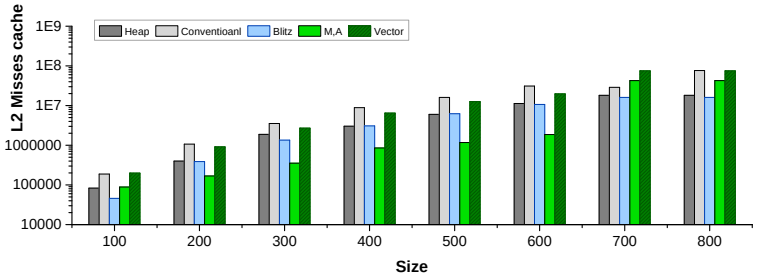


(d) *Temps d'exécution Vector, Blitz++, et multiarray en percentage comparé à CL_array*

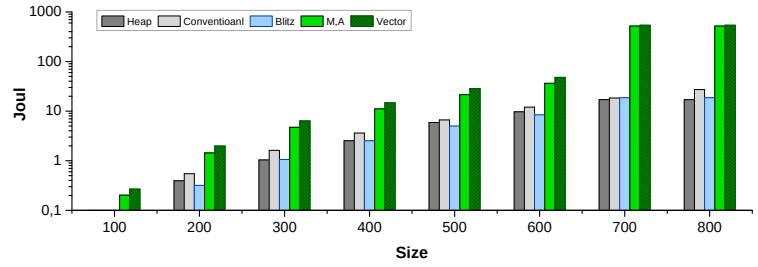
Figure 6.9 – Statistiques obtenues pour l'application contraction tensorielle à deux dimensions



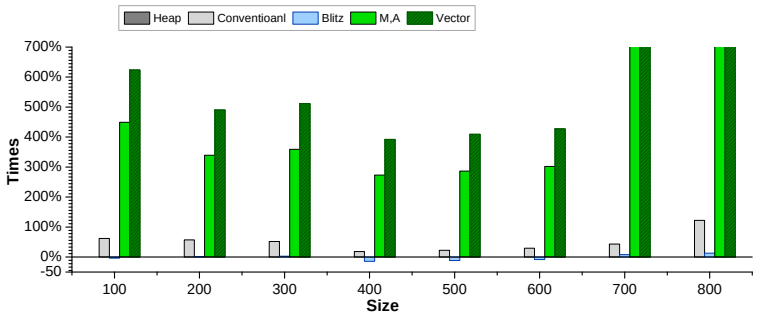
(a) Défauts de caches de niveau trois



(b) Défauts de caches de niveau deux



(c) Consomation d'énergie Vector, Blitz++, multiarray et CL_ARRAY::HEAP en joules



(d) *Temps d'exécution Vector, Blitz++, et multiarray en percentage comparé à CL_array*

Figure 6.10 – Statistiques obtenues pour l'application addition de deux tensors à trois dimensions

6.8 Application "Convolution d'images à deux dimensions"

Une convolution 2D peut être considérée comme un remplaçant de chaque pixel de l'image par la somme pondérée de ses voisins. Le kernel est une autre image, de plus petite taille, qui contient les poids. La somme des poids devrait être égale à un. Si ce n'est pas le cas, la somme des poids peut être calculée au cours de la sommation et utilisée pour évaluer le résultat. pour les besoins de notre évaluation nous avons utilisé le pseudo-code suivant, un format d'images PNG représenté par des matrices à deux dimensions AoS (Array Of Structure) où chaque pixel est représenté par une structure contenant les trois composantes RGB et un kernel aux dimensions 3X3. Nous avons également généré des images pour chaque mesure dont les tailles sont dans l'interval $N \in$ [100X100,200x200,400x400,800x800,1600x1600,3200x3200] pixels, pour chacune des images nous avons mis à jour les attributs pixels RGB par des valeurs alératoires dans l'intervale [0,255].

```
for ( row = 0; row < HEIGHT; row++ ) {
  for ( col = 0; col < WIDTH; col++ ) {
  float accumulation = 0;
  float weightsum = 0;
  for ( i = -1; i <= 1; i++ ) {
    for ( j = -1; j <= 1; j++ ) {
      unsigned char k <- I[row+i, col+j];
      accumulation<-accumulation+ (k * kernel[1+i,1+j]);
      weightsum <- weightsum+kernel[1+i,1+j];
    }
  }
  I2 [row,col]<- size_t(accumulation/weightsum);
}
```

Listing 6.15 – 2D Image Convolution

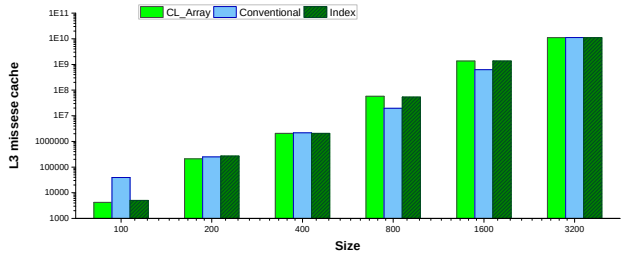
Les statistiques des défaut de cache de niveau deux et trois, l'énergie consommé en joules et les temps d'exécution sont présentés en pourcentages. Ces pourcentages sont obtenus par l'application de la formule suivante $HEAP_{time} - OtherContainer_{time}/HEAP_{time}$, par conséquent les pourcentages positifs indiquent une amélioration tandis que les pourcentages négatifs indiquent une

dégradation.
Comme les librairies boost : :multiarray et vector sont rarement utilisées pour le traitement d'images , ces deux librairies sont exclues de cette évalution. Ainsi, seules les conteneurs conventionnels et ceux Biltz++ sont comparées à CL_ARRAY : :HEAP.
Les statistiques ont été obtenues après l'éxécution du pseudo-code 6.15 et pour chacun des trois conteneurs CL_ARRAY, Conventionnel et Biltz++ repsrentative des conteneurs par index, sur un ordinateur équipé de 32 GO de mémoire, d'un processeur I7 3770, du système d'exploitation linux Debian 14.0.4 et de la version 4.9 du compilateur GCC/G++ avec l'option de compilation O3 . Pour la collecte des statistiques, nous avons utilisé la version 2.8 de la librairie Intel Performance Counter Monitor.

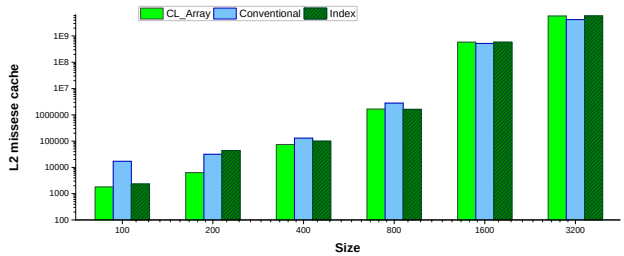
Comme nous pouvons le constater sur les figures 6.11a , 6.11b, 6.11c et 6.11d , pour le cas de la convolution d'images ,

1. les conteneurs Biltz++ qui utilisent une méthode par calcul d'index affichent des performances en termes de statitiques de dafaut de cache L2 et L3, de consomation d'énergie et de temps execution légerement meilleures (de l'ordre de 1 à 3%) que celles constatées pour les conteneurs CL_ARRAY

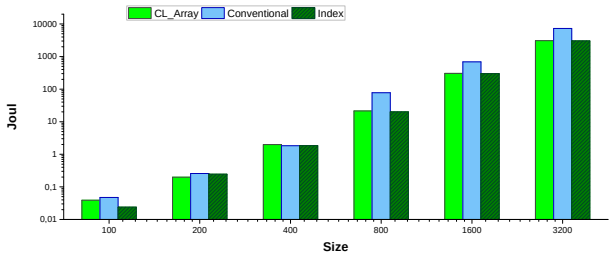
2. Alors que les conteneurs produits par la méthode conventionelle C, affichent des performances très dégradées par rapport à celles constatées pour les deux librairies CL_ARRAY et Biltz++, la dégradation s'explique par la mauvaise localité mémoire qui implique un nombre très élevé des défauts de caches L2 et L3 (voir figures 6.11a et 6.11b)



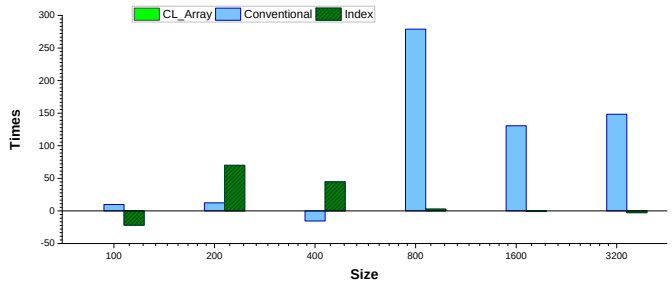
(a) Défauts de caches L3 méthode conventionnelle, et Blitz++ en percentage comparé à CL_ARRAY : :HEAP



(b) Défauts de caches L2 méthode conventionnelle, et Blitz++ en percentage comparé à CL_ARRAY : :HEAP



(c) l'énergie mesurée pour la méthode conventionnelle, et Blitz++ en percentage comparé à CL_ARRAY::HEAP



(d) temps obtenus par la méthode conventionnelle, et Blitz++ en percentage comparé à CL_array::HEAP

Figure 6.11 – Statistiques obtenues pour l'application convolution d'image

6.9 Conclusion

Dans ce chapitre nous avons présenté les techniques d'implémentation qui ont mené à l'élaboration de la librairie CL_ARRAY que nous avons évalué selon sa capacité

1. d'opérer dans des environnements à forte dimensionnalité,
2. de stocker un volume de données très conséquent,
3. de permettre des opérations performantes réputées chronophages de redimensionnement des conteneurs , d'insertion et de suppression des éléments et ce sans consommation mémoire supplémentaire
4. d'optimiser le transfert des données entre entre la mémoire hote et la mémoire des dispositifs OpenCL, et
5. de s'utiliser dans une large variété d'applications d'algèbre multilinéaire et de traitement d'images.

Les résultats obtenus démontrent la pertinence de notre conception ainsi que la bonne conduite de la mise en œuvre. Ils confirment nos prédictions théoriques. Ainsi nous pouvons conclure que la librairie CL_ARRAY contrairement aux autres librairies de l'état de l'art offre un niveau de performance adapté à la fois à la programmation générique, aux calculs parallèles hétérogènes et aux applications de calcul scientifique exigeantes en termes de consommation et de temps d'accès mémoire.

CHAPITRE 7

CONCLUSION GÉNÉRALE

7.1 Bilan

Les travaux présentés dans cette thèse publiés dans [97] s'intéressent aux modèles de programmation parallèles et leurs apports pour les développeurs en termes de performance, d'expressivité et de productivité. Nous avons conçu et implémenté avec succès une nouvelle librairie de conteneurs multidimensionnels pour les compilateurs C++ adaptée à la programmation parallèle hétérogène. Pour l'élaboration de notre librairie CL_ARRAY, nous avons jugé utile de reconsidérer tous les aspects conceptuels liés aux ingénieries des conteneurs, et nous avons proposé une approche conceptualisée, et surtout adaptée pour l'écosystème OpenCL. CL_ARRAY se distingue des autres librairies de l'état de l'art (std : :vector) , boost : :multiarray ou encore Biltz++ par :

- **Un nouveau formalisme** : Au moyen de la sémantique proposée, nous sommes parvenus à formaliser le comportement spatio-temporel des conteneurs.
- **Une simplicité d'utilisation** : la syntaxe présentée aux utilisateurs se caractérise par de simples expressions syntaxiques pour la construction et la manipulation des conteneurs
- **Une portabilité** : la portabilité est un critère fondamental, qui octroit à la nouvelle librairie l'aptitude d'opérer dans différents environnements (systèmes d'exploitation et tout compilateur compatible avec la norme C++11). A cette fin et pour nous assurer de la conformité du critère de portabilité, nous avons évalué la nouvelle librairie sous les systèmes d'exploitation Microsoft avec le compilateur Microsoft Visual Studio et aussi sous Linux Debian utilisant le compilateur G++.
- **Une interopérabilité** : la méthode d'accès standard C/C++ par opérateur subscript [] et OCL_ARRA_VIEW qui implémente une nouvelle sémantique d'itérateurs multidimensionnels confèrent aux conteneurs CL_ARRAY une utilisarion aisée dans des environnements

hétérogènes séquentiels (boost, STL, OpenCV, .etc.) et|ou parallèles (OpenCL , CUDA , OpenMP , pthread , C++ AMP , Intel TBB et MPI)

- **De bonnes Performances** : l'objectif de performance est un enjeu important, les performances que nous avons constatées en matière de temps de construction, des temps d'accès aux conteneurs, des temps de libération mémoires, des consommations d'énergie et mémoires, et enfin des accélérations de transfert des données , confirment l'aptitude de la nouvelle librairie à opérer dans des applications critiques où la performance est requise.

En guise de conclusion , les résultats de ces expérimentation vont dans le sens de notre intuition de départ, à savoir que les modèles métaprogrammés sont les plus à même d'obtenir les meilleures performances possibles mais impliquent des temps de développement très longs alors que les approches objets classique de haut niveau se caractérisent par une meilleure productivité au dépend des performances.

7.2 Pespectives

Les résultats obtenus nous encouragent à prévoir pour un proche futur, la publication de la librairie CL_ARRAY sous License MIT.
Nous mettons en perspectives, l'enrichissement de la libraire CL_ARRAY par d'autres catégories de conteneurs (irrégulier et hétérogène), qui nécessitent pour des raisons d'optimisation la conception d'un nouvel allocateur spécialisé pour garantir une mémoire virtuellement contigüe à l'épreuve des opérations de redimensionnement des insertions et de suppression des éléments qui caractérisent la majorité des conteneurs irréguliers.
Le nouvel allocateur mémoire, permettra de palier à la fois les effets de bords introduits par les allocateurs systèmes (Possibilité de perte de l'allignement mémoire suite aux opérations de réalocation) et garantir un comportement constant indépendamment des alocateurs des systèmes d'exploitation qui devraient en théorie apporter davantages de performance et de flexibilité.

BIBLIOGRAPHIE

[1] The blitz++ library. Online available :http://blitz.sourceforge.net/styled/styled-6/.

[2] Ieee standard for floating-point arithmetic, 2008 . URL https://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf.

[3] Open mpi :open source high performance computing. Online available :http://www.open-mpi.org/.

[4] Opencl 2.1 reference card , rev. 1115. URL https://www.khronos.org/files/opencl21-reference-guide.pdf.

[5] The openmp api specification for parallel programming. Online available :http://openmp.org/wp/.

[6] std : :experimental : :dynarray. Online available :http://en.cppreference.com/w/cpp/container/dynarray.

[7] Cuda nvidia programming guide. Rapport technique, 2009. URL http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.3/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[8] Vulkan 1.0.51 - a specification, Jun 2017. URL https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf.

[9] Badawy Abdel-Hameed, Aggarwal Aneesh, Yeung Donald et Tseng Chau-Wen. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism*, 6(7), 2004.

[10] David Abrahams et Aleksey Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley Professional, 2004.

[11] Bourd Alex. The opencl environment specification, version :2.2, document revision : 06. Rapport technique, Khronos OpenCL Working Group, March 11, 2016.

[12] Andrei Alexandrescu. *Modern C++ Design : Generic Programming and Design Patterns Applied*. Addison Wesley, 2001.

[13] Hux Allen. The opencl extension specification ,version : 2.1 ,document revision : 17. Rapport technique, Khronos OpenCL Working Group, nov 2015. November 5, 2015.

[14] Hux Allen. The opencl extension specification version : 2.2 document revision : 2. Rapport technique, Khronos OpenCL Working Group, feb 2016. February 8, 2016.

[15] AMD. Bolt c++ template library. Online available :http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library/, .

[16] AMD. Opencl optimization guide. Online available :http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/50401315_pgfId-502559, .

[17] AMD. *Accelerated Parallel Processing OpenCL Optimization Guide, Rev 1.0*, chaptitre Unpinned Host Memory, pages 1–6. AMD, 2014.

[18] Roth Amir, Moshovos Andreas et S. Soh Gurindar. Dependence based prefetching for linked data structures. Dans *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[19] Klockner Andreas, Pinto Nicolas, Lee Yunsup, Catanzaro Bryan, Ivanov Paul et Fasih Ahmed. Pycuda and pyopencl : A scripting-based approach to gpu run-time code generation. *Parallel Computing , Volume 38, Issue 3*, 2012.

[20] Sochacki Bartosz. The opencl c++ specification ,version : 1.0 , document revision. Rapport technique, Khronos OpenCL Working Group, apr 2016. April 12, 2016.

[21] Gaster Benedict R. et Howes Lee. The opencl c++ wrapper api, version : 1.2.6, document revision : 02. Rapport technique, Khronos OpenCL Working Group, 2012.

[22] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[23] Alex Bourd. The opencl specification , version : 2.2 , document revision : 06. Rapport technique, Khronos OpenCL Working Group, mar 2016. March 11, 2016.

[24] J. Newburn Chris, So Byoungro, Liu Zhenying, McCool Michael, Ghuloum Anwar, Du Toit Stefanus, Wang Zhi Gang, Hui Du Zhao, Chen Yongjian, Wu Gansha, Guo Peng, Liu Zhanglin et Zhang Dan. Intel array building blocks :a retargetable, dynamic compiler and embedded language. Dans *CGO 11 Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011.

[25] Lawrence Crowl et Matt Austern. C++ dynamic arrays. Online available :http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2648.html.

[26] L. Dagum et R. Menon. Openmp : an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998. ISSN 1070-9924.

[27] Usman Dastgeer et Christoph Kessler. Smart containers and skeleton programming for gpu-based systems. *International Journal of Parallel Programming, Volume 44, Issue 3*, 2016.

[28] Beman Dawes, Douglas Gregor, Jeremiah Willcock et Andrew Lumsdaine. Concepts for the c++0x standard library : Numerics (revision 2). Rapport technique, Programming Language C++, Library Working Group, 2008.

[29] Demidov Denis, Ahnert Karsten, Rupp Karl et Gottschling Peter. Programming cuda and opencl : A case study using modern c++ libraries. *SIAM J. SCI. COMPUT*, 35(5):453–472, 2013.

[30] N. Drosinos et N. Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. *18th Int. Parallel and Distributed Symposium,*, 2004.

[31] D. Berger Emery, G. Zorn Benjamin et S. McKinley Kathryn. Reconsidering custom memory allocation. Dans *OOPSLA 02 Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2002.

[32] Loiic Fejoz. *Developpement prouve de structures de donnees sans verrou*. Thèse de doctorat, Henri Poincarie-Nancy-University, 2008.

[33] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*, chaptitre Strategy, pages 349–359. Pearson Education, eBook Format, 1994.

[34] Ronald Garcia et Andrew Lumsdaine. Multiarray : a c++ library for generic programming with arrays. *Software : Practice and Experience, Volume 35, Issue 2*, 2005.

[35] Ronald Garcia, Jeremy Siek et Andrew Lumsdaine. Boost.multiarray. Online available :http://www.boost.org/doc/libs/1_57_0/libs/multi_array/doc/index.html, 2000-2001.

[36] Benedict R. Gaster. The opencl c++ wrapper api, version : 1.1 , document revision :04. Rapport technique, Khronos Group Inc, 2010.

[37] Benedict R Gaster et Lee Howes. Opencl c++. Dans *GPGPU 6 Proceedings of 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013.

[38] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry et Dana Schaa. *Heterogeneous Computing with OpenCL, 2nd Edition Revised OpenCL 1.2 Edition*, chaptitre Data Management, pages 147–159. Elsevier Inc, Waltham USA, 2011.

[39] Fayez Gebali. *ALGORITHMES AND PARALLEL COMPUTING*. WILLEY, 2011.

[40] DAVIDE DI GENNARO. *ADVANCED C++ METAPROGRAMMING*. Apress, 2012.

[41] Edwards H. Carter, Daniel Sunderland, Vickia Porter, Chris Amsler et Sam Mish. Manycore performance-portability : Kokkos multidimensional array library. *Scientific Programming Volume 20, Issue 2*, pages 89–114, 2012.

[42] Johnson M. Hart. *Windows System , Fourth Edition*, chaptitre Source Code Portability : Windows, UNIX, and Linux, pages 549–555. Massachusetts USA, Addison-Wesley, 2010.

[43] Lee Howes et Maria Rovatsou. Sycltm specification, sycl integrates opencl devices with modern c++ version 1.2. Rapport technique, Khronos Working Group, 2015. Revision Date : 2015-05-08.

[44] Intel. Intel threading building blocks (intel tbb) 4.3 update 3. Online available :https://www.threadingbuildingblocks.org.

[45] Lee Jaekyu, Kim Hyesoon et Vuduc Richard. When prefetching works, when it doesnt, and why. *ACM Transactions on Architecture and Code Optimization, ,*, Vol. 9 , Issue 1,(No. 2), mar 2012.

[46] Collins Jamison, Sair Suleyman, Calder Brad et M. Tullsen Dean. Pointer cache assisted prefetching. Dans *In Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO-35)*, nov 2002.

[47] Google John Kessenich et Intel Boaz Ouriel. Spir-v specification, May 2017. URL https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf.

[48] Nicolai M. Josuttis. *The C++ Standard Library, A Tutorial and Reference, 2nd Edition*, chaptitre Supplementary Chapter, pages 114–1147. addison-wesley, 2012.

[49] Nicolai M. Josuttis. *The C++ Standard Library, A Tutorial and Reference, 2nd Edition*. addison-wesley, Michigan , USA, 2012.

[50] Azharul Hasan K. M, Islam Kamrul, Islam Mojahidul et Tsuji Tatsuo. An extendible data structure for handling large multidimensional data sets. Dans *Proceedings of the 12th International Conference on Computer and Information Technology (ICCIT 2009), Dhaka, Bangladesh*, 2009.

[51] Amir Kamil, Yili Zheng et Katherine Yelick. A local-view array library for partitioned global address space c++ programs. Dans *ARRAY 14 Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2014.

[52] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones et Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. Dans *ICFP 10 Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, 2010.

[53] David Kieras. Using c++11 smart pointers. EECS Department, University of Michigan, June 2016.

[54] David Kirk et Wen-mei Hwu. *Programming Massively Parallel Processors*, chaptitre A Brief Introduction to OpenCL, pages 205–220. Elsevier, Burlington USA, 2010.

[55] Adam Lake. Getting the most from opencl 1.2 , how to increase performance by minimizing buffer copies on intel processor graphics. Rapport technique, Intel, 2014.

[56] Howes Lee. The opencl specification, version : 2.1, document revision : 23. Rapport technique, Khronos OpenCL Working Group, nov 2015. November 11, 2015.

[57] Xiaolei Li, Jiawei Han et Hector Gonzalez. High-dimensional olap : A minimal cubing approach. Dans *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, 2004.

[58] Numerics library. Online available :http://www.open-std.org/jtc1/sc22/open/n2356/lib-numerics.html.

[59] Mendakiewicz Lukasz et Sutter Herb. Multidimensional bounds, index and array view. Rapport technique, Microsoft Corporation, 2014.

[60] Kyle Lutz. Boost.compute. Online available :http://kylelutz.github.io/compute/, 2014.

[61] Mernik Marjan, Heering Jan et M.Sloane Anthony. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[62] Nir Shavit Maurice Herlihy. *The Art of Multiprocessor ..Programming..* Elsevier, 2012.

[63] Michael McCool, James Reinders et Arch Robison. *Structured Parallel Patterns for Efficient Computation*. Elsevier, Inc., Waltham, USA, 2012.

[64] James Reinders Michael McCool, Arch D. Robison. *Structured Parallel Programming , Patterns for Efficient computation*. Elsevier, 2012.

[65] Microsoft. C++ amp (c++ accelerated massive parallelism). Online available :https://msdn.microsoft.com/en-us/library/hh265137.aspx.

[66] Microsoft. C++ amp : Language and programming model version 1.2. Rapport technique, Microsoft Corporation, 2013.

[67] Aaftab Munshi. The opencl specification, version :1.1, document revision :44. Rapport technique, The Khronos Group Inc, 2011.

[68] Aaftab Munshi. The opencl specification, version : 1.2, document revision : 19. Rapport technique, The Khronos Group Inc, 2012.

[69] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung et Dan Ginsburg. *OpenCL Programming Guide*, chaptitre Programming with OpenCL C, pages 146–147. Adisson-Wesly, Upper Saddle River, NJ Boston , Indianapolis , San Francisco, 2011.

[70] Aaftab Munshi et Lee Howes. The opencl specification, version : 2.0, document revision 26. Rapport technique, The Khronos Group Inc, 2014.

[71] Park Neungsoo, Hong Bo et Fellow Viktor K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, VOL. 14, NO. 7, jul 2003.

[72] Nvidia. Thrust. Online available :http://docs.nvidia.com/cuda/thrust/.

[73] NVIDIA. Nvidia opencl best practices guide version 1.0. Rapport technique, Nvidia, 2009.

[74] Semih Okur et Danny Dig. How do developers use parallel libraries ? Dans *FSE 12 Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[75] Peter Pacheco. *An introduction to PARALLEL PROGRAMMING*. Elsevier, 2011.

[76] S. Rakic Predrag, Stricevic azar et Rakic Zorica Suvajdzin. Statically typed matrix : in c++ library. Dans *BCI 12 Proceedings of the Fifth Balkan Conference in Informatics*. ACM, 2012.

[77] Banger Ravishekhar et Bhattacharyya Koushik. *OpenCL Programming By Example*, pages 35–58. PACKT PUBLISHING, Birmingham UK, 2013.

[78] Tay Raymond. *OpenCL Parallel Programming Development Cookbook.*, chaptitre Understanding how to solve SpMV using VexCL, pages 216–219. PACKT Publiching, Birmingham UK., 2013.

[79] James Reinders. *Intel Threading Building Blocks, Outfitting C++ for Multi-Core Processor Parallelism*, chaptitre Containers, pages 80–91. OREILLY, 2007.

[80] Maria Rovatsou. Sycltm provisional specification, sycl integrates opencl devices with modern c++ using a single source design version 2.2. Rapport technique, Khronos OpenCL Working Group, 2016. Revision Date : 2016/02/15.

[81] Boris Schnaling. *The Boost C++ Libraries, A quickstart introduction with over 250 examples*. XML PRESS, 2011.

[82] Mehrotra Sharad et Harrison Luddy. Examination of a memory access classification scheme for pointer-intensive and numeric programs. Dans *the 10th International Conference on Supercomputing, (Philadelphia,PA), ACM*, 1996.

[83] J. E. Stone, D. Gohara et G. Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010. ISSN 1521-9615.

[84] Bjarne Stroustrup. Evolving a language in and for the real world : C++ 1991-2006. Dans *HOPL III Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2006.

[85] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesely, Michigan USA, 2013.

[86] Gabriel Tanase, Antal A. Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato et Lawrence Rauchwerger. The stapl parallel container framework. Dans Calin Cascaval et Pen-Chung Yew, éditeurs, *PPOPP*, pages 235–246. ACM, 2011. ISBN 978-1-4503-0119-0. URL http://dblp.uni-trier.de/db/conf/ppopp/ppopp2011.html#TanaseBFHPPSTXMVBAR11.

[87] Stefan Van Der Walt, S. Chris Colbert et Gaël Varoquaux. The numpy array : A structure for efficient numerical computation. *Computing in Science and Engineering, volume 13, issue 2*, 2011.

[88] David Vandevoorde et M. Nicolai Josuttis. *C++ Templates : The Complete Guide*. Addison Wesley, 2002.

[89] Todd L. Veldhuizen. Arrays in blitz++,. Dans *ISCOPE98, In Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*, 1998.

[90] VexCL. C++ vector expression template library for opencl/cuda. Online available :http://ddemidov.github.io/vexcl/index.html.

[91] William von Hagen. *The Definitive Guide to GCC, Second Edition*. APRESS, 2007.

[92] ANTHONY WILLIAMS. *C++ Concurrency in Action, PRACTICAL MULTITHREADING*. MANNING SHELTER ISLAND, 2012.

[93] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm et John Melonakos. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015. URL https://github.com/arrayfire/arrayfire.

[94] Xin Yan, Xiaohua Shi, Lina Wang et Haiyan Yang. An opencl micro-benchmark suite for gpus. *J Supercomput Springer Science*, 69(2):693–713, 2014.

[95] Paek Yunheung, Hoeflingert Jay et Paduat David. Simplification of array access patterns for compiler optimizations montreal. canada acm. Dans *PLDI 98, Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998.

[96] Yili Zheng, Amir Kamil, Michael B. Driscoll et Hongzhang Shan. Upc++ : A pgas extension for c++. Dans *IPDPS 14, The 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

[97] Chakib Mustapha Anouar Zouaoui et Nasreddine Taleb. Cl_array : A new generic library of multidimensional containers for c++ compilers with extension for opencl framework. *Computer Languages, Systems & Structures*, 50:53 – 81, 2017. ISSN 1477-8424. URL http://www.sciencedirect.com/science/article/pii/S147784241630135X.

Abstract:

This doctoral thesis presents a new metaprogramming library, CL_ARRAY, which offers multiplatform and generic multidimensional data containers for C++ specifically adapted for parallel programming. The CL_ARRAY containers are built around a new formalism for representing the multidimensional nature of data as well as the semantics of multidimensional pointers and contiguous data structures. We also present OCL_ARRAY_VIEW, a concept based on metaprogrammed enveloped objects that supports multidimensional transformations and multidimensional iterators designed to simplify and formalize the interfacing process between OpenCL APIs, standard template library (STL) algorithms and CL_ARRAY containers. Our results demonstrate improved performance and energy savings over the three most popular container libraries available to the developer community for use in the context of multi-linear algebraic applications.

Résumé :

Cette thèse présente une nouvelle conception de conteneurs de données multidimensionnels et multiplateforme, particulièrement adaptée pour l'écosystème OPENCL, qui a permis l'élaboration d'une nouvelle librairie que nous baptisons CL_ARRAY. Outre, la simplicité d'utilisation, une sécurité intégrée et une sémantique à typage statique qui caractérisent les conteneurs du CL_ARRAY, nous proposons un concept similaire au ARRAY_VIEW du C++AMP pour formaliser le modèle d'exécution et ou le modèle dépendances à faible grains , par ailleurs et même si les autres librairies qui proposent des conteneurs évolués , ne sont pas adaptées pour l'écosystème OPENCL, nos résultats ont démontré une performance supérieure et une économie mémoire conséquente comparativement à trois librairies de conteneurs les plus populaires auprès de la communauté des développeurs dans le contexte d'applications de l'algèbre multilinéaire.

ملخص:

تقدم أطروحة الدكتوراه هاته مكتبة جديدة للميتابروغرام ، CL_ARRAY ، والتي تقدم حاويات معطيات متعددة الأبعاد عامة و متعددة المناهج لـC++ خصيصا للبرمجة المتوازية .تم بناء حاويات ال CL_ARRAY تبعا لصيغة جديدة لتمثيل الطبيعة المتعددة الأبعاد للمعطيات و أيضا تبعا لدلالة المؤشرات المتعددة الأبعاد وهياكل المعطيات المتصلة .ونقدم أيضا OCL_ARRAY_VIEW ، وهو مفهوم يتأسس على عناصر برمجية بالميتابروغرام مغلفة و التي تدعم التحولات والمكررات المتعددة الأبعاد و التي صممت لتبسيط وترميز عملية التواصل بين واجهات برمجة OpenCL ، خوارزميات المكتبة القاعدية STL و حاويات ال CL_ARRAY .نتائجنا أثبت تحسين الأداء وتوفير الطاقة على ثلاثة مكتبات الحاويات الأكثر شعبية و المتاحة لجماعة المبرمجين لاستخدامها في سياق تطبيقات جبري متعددة الخطية.